普通高等教育“计算机类专业”规划教材

Python程序设计

董付国　编著

清华大学出版社

普通高等教育"计算机类专业"规划教材

Python 程序设计

董付国　编著

清 华 大 学 出 版 社
北　京

内 容 简 介

全书共两篇17章，第一篇介绍Python数据类型、控制结构、正则表达式、类与函数设计、文件操作、异常处理与程序调试等内容。第二篇通过大量案例介绍Python在GUI编程、图形图像编程、音乐编程与语音识别、科学计算可视化、网络编程、逆向工程与软件分析、大数据处理、Windows系统编程等方面的应用。

本书可以作为计算机科学与技术、数字媒体技术、软件工程、网络工程、信息安全、会计、经济、金融、心理学、统计等专业本科和研究生"Python程序设计"课程教材和具有一定Python基础的读者进阶学习资料，多领域Python应用开发人员以及打算使用Python快速实现研究思路和创意的科研人员和管理人员的参考书，而且也适合打算学习一门快乐的编程语言并编写几个小程序来娱乐的读者。

图书在版编目(CIP)数据

Python程序设计/董付国编著. —北京：清华大学出版社，2015
普通高等教育"计算机类专业"规划教材
ISBN 978-7-302-40723-2

Ⅰ. ①P…　Ⅱ. ①董…　Ⅲ. ①软件工具—程序设计—高等学校—教材　Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2015)第151726号

责任编辑： 白立军
封面设计： 傅瑞学
责任校对： 白　蕾
责任印制： 何　芊

出版发行： 清华大学出版社
网　　址： http://www.tup.com.cn，http://www.wqbook.com
地　　址： 北京清华大学学研大厦A座　　**邮　　编：** 100084
社 总 机： 010-62770175　　**邮　　购：** 010-62786544
投稿与读者服务： 010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈： 010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载： http://www.tup.com.cn，010-62795954
印 刷 者： 三河市君旺印务有限公司
装 订 者： 三河市新茂装订有限公司
经　　销： 全国新华书店
开　　本： 185mm×260mm　　**印　　张：** 19　　**字　　数：** 460千字
版　　次： 2015年8月第1版　　**印　　次：** 2015年8月第1次印刷
印　　数： 1～2000
定　　价： 39.00元

产品编号：064549-01

Python 是由 Guido van Rossum 于 1989 年底发明的，第一个公开发行版发行于 1991 年。Python 推出不久便迅速得到各行业人士的青睐，目前已经渗透到计算机科学与技术、统计分析、移动终端开发、科学计算可视化、逆向工程与软件分析、图形图像处理、人工智能、游戏设计与策划等几乎所有专业和领域，并且已经成为卡耐基·梅隆大学、麻省理工学院、加州大学伯克利分校、哈佛大学等国外很多大学计算机专业和非计算机专业的程序设计入门教学语言，国内也有不少学校陆续开设了 Python 程序设计课程。Python 语言连续多年在 TIOBE 网站的编程语言排行榜上排名七八位左右，并于 2011 年 1 月被 TIOBE 网站评为 2010 年年度语言。在 2014 年 12 月份 IEEE Spectrum 推出的编程语言排行榜中，Python 更是取得了第 5 位的好名次。

Python 是一门开源的高级动态编程语言，支持命令式编程、函数式编程、面向对象程序设计，语法简洁清晰，并且拥有大量功能丰富而强大的标准库和扩展库，可以帮助各领域的科研人员或策划师，以及管理人员快速实现并验证自己的思路与创意。Python 使得用户可以把主要精力放在业务逻辑的设计与实现上，而不用过多考虑语言本身的细节，开发效率非常高，其精妙之处令人赞叹。另外，还可以使用 py2exe 工具将 Python 程序轻易转换为 exe 可执行程序，它脱离 Python 解释器以便在没有安装 Python 环境的 Windows 平台运行，极大地方便了用户的使用。

Python 是一门快乐的语言，学习和使用 Python 也是一个快乐的过程。与 C 语言系列或 Java 等语言相比，Python 更容易学习和使用，但这并不意味着可以非常轻松愉快地掌握 Python。熟练掌握和运用 Python 仍需要通过大量的练习来锻炼自己的思维和熟悉 Python 编程模式，同时还需要经常关注 Python 社区优秀的代码以及各种扩展库的动态。

Python 是一门优雅的语言。如果您有其他程序设计语言的基础，那么在学习和使用 Python 的时候，一定不要把其他语言的编程习惯带到 Python 中来。您应该尽量尝试从最自然、最简洁的角度出发去思考和解决问题，这样才能写出更加优雅、更加 Pythonic 的代码。

本书内容组织

全书共两篇 17 章，第一篇重点介绍和讲解 Python 程序设计基础知识，主要包括 Python 基本数据结构、控制结构、正则表达式、类与函数设计、文件操作、异常处理与程序调试等内容。第二篇通过大量的案例介绍 Python 的多个扩展库在 GUI 编程、图形图像编程、音乐编程与语音识别、科学计算可视化、网络编程、逆向工程与软件分析、大数据处理、多语言混合编程和 Windows 系统编程等方面的应用。

在真正编写和开发一个应用程序时，需要用到方方面面的知识，既包括 Python 基本语法与数据结构，又包括选择、循环等控制语句，类与函数设计，异常处理结构，文件处理，以及

各种扩展库的综合运用,甚至还有可能需要与其他语言结合,这些知识总是互相交织在一起的,很难彻底分开。因此,在编写本书前面的内容时,可能会偶尔用到后面的知识,这实在是一个很难避免的问题。虽然作者已经很努力地把内容组织为容易理解的形式,但不得不说,仍有很多知识需要您前后翻阅多次才能真正融会贯通。

本书作者具有 15 年的程序设计教学经验,讲授过多门程序设计语言,分别使用汇编语言、C、C++、C#、Java、PHP、Python 等不同语言编写过大量的应用软件。在本书内容的组织和安排上,结合了自己多年的应用开发和教学工作中积累的许多案例,把实际应用中的大量案例巧妙地糅合进了相应的章节。

本书内容组织的最大特点是不仅信息量大,而且知识点全面、密集。考虑到 Python 软件和扩展库的安装过程较为简单,绝大部分读者都能够顺利安装,在书中花费大量篇幅一步步介绍安装过程的意义和必要性并不大。因此,在整本书中都没插入任何软件和相关扩展库的安装过程截图,而是充分利用有限的篇幅来介绍和讲解知识点,可以说是物超所值。

本书适用读者

本书不仅可以作为 Python 程序设计语言课程的教材和具有一定 Python 基础的读者的进阶学习资料,还可以作为多个领域的 Python 应用开发人员的参考书,既可以作为计算机科学与技术、数字媒体技术、软件工程、网络工程、信息安全、会计、经济、金融、心理学、统计等多个专业本科和研究生的程序设计教材,也可以作为打算使用 Python 快速实现自己研究思路和创意的科研人员及管理人员的参考书,当然也适合那些打算利用业余时间学习一门快乐的程序设计语言并编写几个小程序来娱乐的读者。

如果作为本科专业课程教材,建议学时为 48 学时课堂授课+16 学时上机实验,如果采用边讲边练的教学模式,建议控制在 72 学时左右;如果作为非计算机专业研究生教材,建议为 48 学时课堂授课,上机部分可以由研究生自行完成。除了讲授第一篇中全部知识以外,根据学生基础、专业特点和培养目标,再酌情选择第二篇中的部分章节进行讲解。另外,第一篇的知识应尽量按照本书组织的先后顺序进行讲解和学习,而第二篇中的章节可根据需要进行前后调整,并不一定要严格按照本书的顺序。如果作为本科非计算机专业程序设计语言公共课或选修课教材,建议采用 48 学时边讲边练的教学模式,可以略过第一篇中的高级话题以及第 8 章的内容,再根据具体的学生专业选择第二篇中的一到两章进行讲解,其余章节可以由学生根据自己的兴趣进行阅读。

教学资源

本书提供全套教学课件、源代码、课后习题答案与分析以及授课计划和学时分配表,配套资源可以登录清华大学出版社官方网站(www.tup.com.cn)下载或与作者联系索取,作

者电子邮箱为 dongfuguo2005@126.com。

由于时间仓促，作者水平有限，书中难免出现错误和不足之处，还请同行指正并通过电子邮件等方式进行反馈，作者将不定期在 QQ 空间和微信发布及更新勘误表。

感谢

首先感谢父母对我的养育之恩，在当年那么艰苦的条件下还坚决支持我读书，而没有让我像其他同龄孩子一样辍学。感谢姐姐、姐夫多年来对我的爱护以及在老家对父母的照顾；感谢善良的弟弟、弟媳在老家对父母的照顾，正是有了你们，我才能在远离家乡的城市安心工作。感谢我的妻子在生活中对我的大力支持，也感谢懂事的小女儿在我工作的时候能够在旁边安静地读书而尽量不打扰我，并在定稿前帮我检查出了几个错别字和一个错误的序号。

感谢本书定稿前的第一批读者，山东工商学院数字媒体技术专业(服务外包方向)2012级的毛玉婷同学和巩晓同学，认真阅读了全书并检查其中的错别字。当然，也感谢每一位读者，感谢您在茫茫书海中选择了本书，并衷心祝愿您能够从本书中受益，学到您需要的知识！

本书的出版获 2014 年山东省普通高校应用型人才培养专业发展支持计划项目资助。我校专业共建合作伙伴——浪潮优派科技教育有限公司总裁邵长臣先生审阅了全书，并提出很多宝贵的意见，在此致以诚挚的谢意。本书在编写出版过程中得到了清华大学出版社的大力支持和帮助，在此表示衷心的感谢。

董付国定稿于山东烟台
2015 年 5 月

第一篇　Python 基础

第二篇 Python 高级编程与应用

第一篇　Python 基础

毫无疑问，对于 Python 程序员来说，能够熟练运用各种扩展库非常重要，使用优秀、成熟的扩展库可以帮助人们快速实现自己的业务逻辑和创意。但也必须清楚地认识到，Python 语言基础知识和基本数据结构的熟练掌握是理解和运用其他扩展库的必备条件。本书第一篇详细介绍列表、元组、字典、字符串等基本数据结构，正则表达式，选择、循环等主要控制结构，函数设计与面向对象程序设计，文本文件与二进制文件操作以及异常处理结构与 Python 程序调试技术。为了更好地演示一些基础知识在实际开发中的运用，有时候不得不在前面用到后面介绍的知识，如果遇到这种情况，您可以暂时不去考虑太多细节，当然也可以翻阅后面的内容或者查阅相关文档来帮助理解。

第 1 章　基 础 知 识

Python 是一门跨平台的开源、免费的解释型脚本语言，同时也支持伪编译以进行优化和提高运行速度，还支持使用 py2exe 工具将 Python 程序转换为 exe 可执行程序以使得可以在没有安装 Python 解释器和相关依赖包的平台上运行；Python 同时支持命令式编程、函数式编程和面向对象的编程，语法简洁清晰，并且拥有大量的几乎支持所有领域应用开发的成熟扩展库；最后，Python 就像胶水一样，可以把多种不同语言编写的程序融合到一起实现无缝拼接，更好地发挥不同语言和工具的优势。

1.1　Python 语言版本之争

众所周知，Python 目前同时发行 Python 2. x 和 Python 3. x 两个不同系列的版本，并且互相之间不兼容，在本书开始编写的时候，最新版本分别为 Python 2. 7. 8 和 Python 3. 4. 2，本书编写完成时最新版本分别为 Python 2. 7. 9 和 Python 3. 4. 3。对于很多初级用户而言，最纠结的一个问题很可能是自己到底应该选择哪个版本，是选择 Python 2. x 还是 Python 3. x 呢？是选择 Python 2. 7. x 还是 Python 2. 6. x 呢？对于 Python 的版本演化历史，这里不多解释，需要说明的是，并不是数字越大表示版本越新，例如 Python 2. 7. 9 比 Python 3. 3 晚几个月发行。另外，虽然同系列的版本中高版本比低版本更加完善和成熟，但这并不意味着最新的才是最适合您的。这是因为很多扩展库的发行总是滞后于 Python 发行的版本，甚至目前还有很多扩展库不支持 Python 3。因此，在选择 Python 的时候，一定要先考虑清楚自己学习 Python 的目的是什么，打算做哪方面的开发，有哪些扩展库可用，这些扩展库最高支持哪个版本的 Python，是 Python 2. x 还是 Python 3. x，最高支持到 Python 2. 7. 6 还是 Python 2. 7. 9。这些问题都确定以后，再做出自己的选择，这样才能事半功倍，而不至于把太多时间浪费在 Python 以及扩展库的反复安装和卸载上。同时还应该注意，当更新的 Python 版本推出之后，不要急于更新，而是应该确定自己所必须使用的扩展库也推出了较新版本之后再一起进行更新。

尽管如此，Python 3. x 毕竟是大势所趋，如果您暂时还没想到要做什么行业领域的应用开发，或者仅仅是为了尝试一种新的、好玩的语言，那么请毫不犹豫地选择 Python 3. x 系列的最高版本（目前是 Python 3. 4. 3）。作者也相信，越来越多的扩展库将会在短时间内推出支持 Python 3. x 的版本。

安装好 Python 之后，在“开始”菜单中启动 IDLE（Python GUI）即可启动 Python 解释器并可以看到当前安装的 Python 版本号，如图 1-1 和图 1-2 所示。当然，也可以启动 Python（command line）来开始美妙的 Python 之旅。在 IDLE（Python GUI）和 Python（command line）两种界面中，都以 3 个大于号>>>作为提示符，可以在提示符后面输入要执行的语句。在本书给出的示例代码中，>>>符号不需要输入，仅表示该代码是在交互式方式下运行，而不带有该提示符的代码则表示是以脚本程序的方式运行的。本书主要使用

IDLE(Python GUI)来介绍 Python 程序的开发与应用。

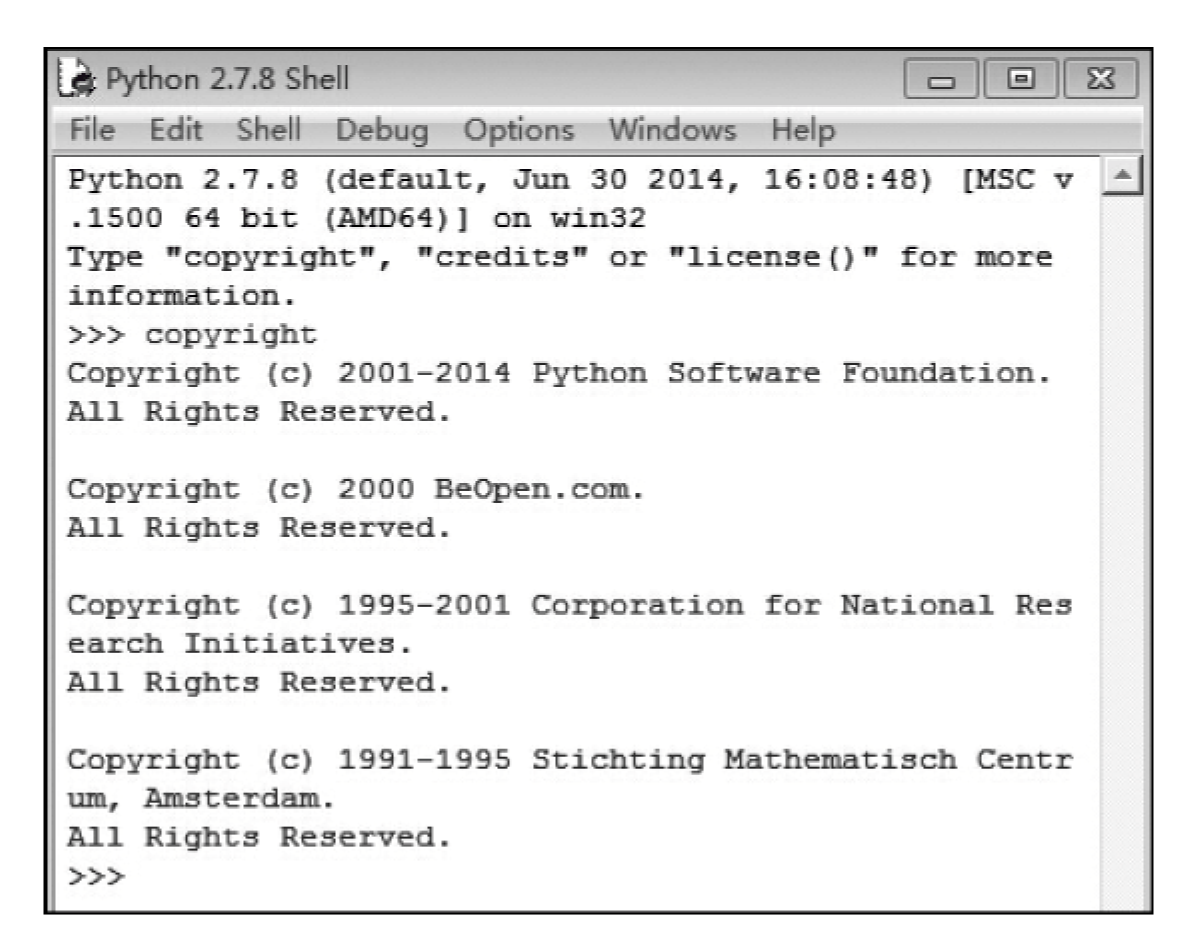


图 1-1　Python 2.7.8 主界面

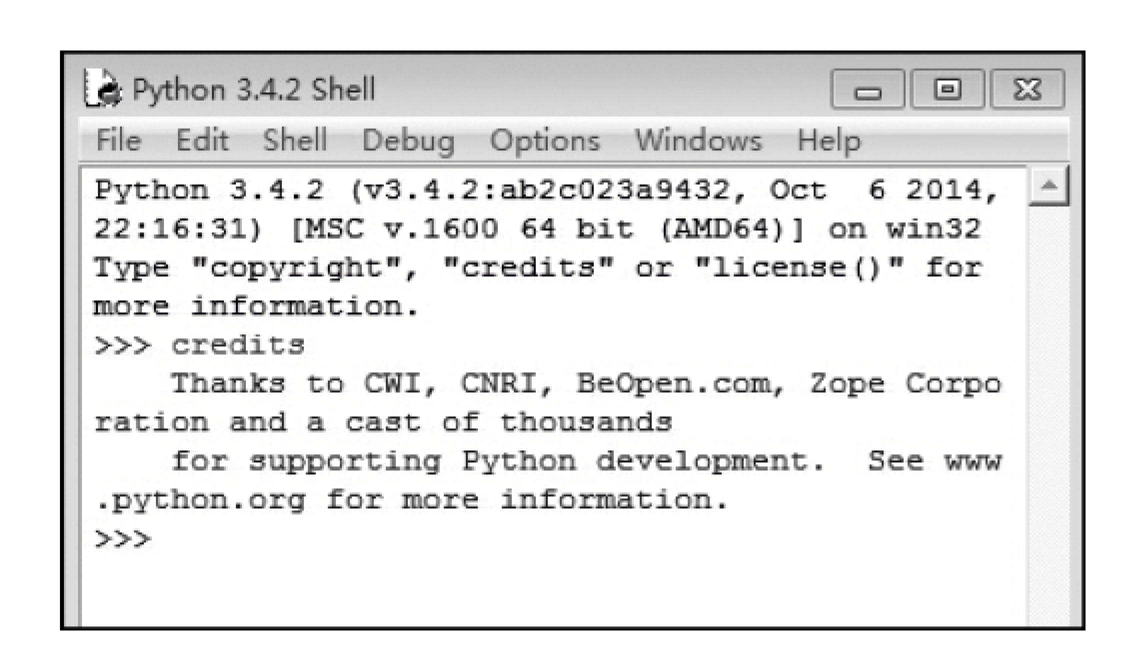


图 1-2　Python 3.4.2 主界面

除了在启动主界面上查看已安装的 Python 解释器版本之外，还可以使用命令进行查看，例如：

```
>>> import sys
>>> sys.version
'3.4.2 (v3.4.2:ab2c023a9432, Oct 6 2014, 22:16:31) [MSC v.1600 64 bit (AMD64)]'
>>> sys.winver
'2.7'
>>> sys.version_info
sys.version_info(major=2, minor=7, micro=8, releaselevel='final', serial=0)
```

有时候可能需要同时安装多个不同的版本，例如同时安装 Python 2.7.8 和 Python 3.4.2，并根据不同的开发需求在两个版本之间进行切换。如果无法正常运行程序的话，可以在调用 Python 主程序时指定其完整路径，或者通过修改系统 Path 变量来实现不同版本之间的切换。在 Windows 7 系统下修改系统 Path 变量的步骤如下：单击“开始”菜单，在弹出的菜单中右击“计算机”并选择“属性”，单击“高级系统设置”切换至“高级”选项卡，单击“环境变量”，然后修改系统 Path 变量中 Python 的安装路径，如图 1-3 所示。



图 1-3 Windows 7 环境中系统 Path 变量的修改方法

1.2 Python 安装与简单使用

关于 Python 的安装步骤这里就不再赘述了，它的安装与大多数软件的安装并没什么明显的不同，打开 Python 官方主页 https://www.python.org/，然后选择适合自己的版本下载安装即可。如果使用 Linux 系统，例如 Ubuntu，那么很可能已经预装了某个版本的 Python，请根据需要进行升级。未经特别说明的话，本书所有示例均在 Windows 7 平台上进行开发和演示。

安装好以后，默认以 IDLE 为开发环境，当然也可以安装使用其他开发环境。本书均以 IDLE 为例，如果使用交互式编程模式，那么直接在 IDLE 提示符(>>>)后面输入相应的命令并回车执行即可，如果执行正常的话，马上就可以看到执行结果。例如：

```
>>> 3+5
8
>>> import math
>>> math.sqrt(9)
3.0
>>> 3* (2+6)
24
```

一般来讲，用户可能更需要编写 Python 程序来实现特定的业务逻辑，同时也方便代码的不断完善和重复利用，毕竟直接使用交互编程模式不是很方便，此时可以在 IDLE 界面中

执行 File→New File 命令创建程序文件，输入程序并保存为文件(务必要保证扩展名为 py)后，执行 Run→Run Module 命令运行，程序运行结果将直接显示在 IDLE 交互界面上。除此之外，也可以通过在资源管理器中双击扩展名为 py 的 Python 程序文件来运行；在有些情况下，可能还需要您在命令提示符环境中运行 Python 程序文件。在“开始”菜单的“附件”中单击“命令提示符”，然后执行 Python 程序。例如，假设有程序 helloworld. py，其内容如下：

```
def main():
    print('Hello world')
main()
```

在 IDLE 环境中运行该程序后，结果如图 1-4 所示。在命令提示符环境中运行该程序后，结果如图 1-5 所示，图中演示了两种执行 Python 程序的方法，虽然第二种方法看上去更加简单，但是请尽量使用第一种方法来运行 Python 程序。

```
>>> ================================ RESTART ========
=
>>>
Hello world
>>>
```

图 1-4　在 IDLE 中运行程序

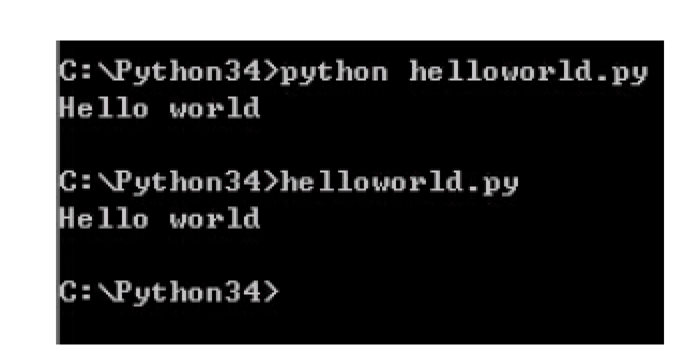


图 1-5　在命令提示符中运行程序

在实际开发中，如果能够熟练使用集成开发环境 IDLE 提供的一些快捷键，将会大幅度提高编写速度和开发效率。在 IDLE 环境下，比较常用的快捷键如表 1-1 所示。

表 1-1　IDLE 环境下常用快捷键

快捷键	功能说明
Alt+P	浏览历史命令(上一条)
Alt+N	浏览历史命令(下一条)
Ctrl+F6	重启 Shell，之前定义的对象全部无效
F1	打开 Python 帮助文档
Alt+/	自动补全前面曾经出现过的单词，在多个单词中循环选择
Ctrl+]	缩进代码块
Ctrl+[	取消代码块缩进
Alt+3	注释代码块
Alt+4	取消代码块注释

1.3　使用 pip 管理扩展库

目前，pip 已经成为管理 Python 扩展库(或模块，一般不做区分)的主流方式，大多数扩展库都支持这种方式进行安装、升级、卸载等操作，使用这种方式管理 Python 扩展库只需要在保证计算机联网的情况下输入几个命令即可完成，极大地方便了用户。

在 Python 2.7.9 和 Python 3.4.0 之前，需要首先安装 pip 命令，而在 Python 2.7.9 以及 Python 3.4.0 之后的安装包中则已经集成了该命令。在 Python 2.7.9 和 Python 3.4.0 之前，首先从 https://pypi.python.org/pypi/pip 下载文件 get-pip.py，然后在命令提示符下执行命令：

```
python get-pip.py
```

即可自动完成 pip 的安装。当然，需要保证计算机处于联网状态。

安装完成以后，就可以在命令提示符下使用 pip 来完成扩展库的安装、升级、卸载等操作，pip 常用命令的使用方法如表 1-2 所示。

表 1-2 pip 常用命令的使用方法

pip 命令示例	说　明
pip install SomePackage	安装 SomePackage 模块
pip list	列出当前已安装的所有模块
pip install --upgrade SomePackage	升级 SomePackage 模块
pip uninstall SomePackage	卸载 SomePackage 模块

1.4 Python 基础知识

本节重点介绍 Python 语言基础知识，包括对象模型、变量、运算符与表达式、内置函数以及数字、字符串等内置基本数据类型等。这些基础知识对于后续知识的理解非常重要，建议读者掌握和理解。

1.4.1 Python 对象模型

对象是 Python 语言中最基本的概念之一，在 Python 中的一切都是对象，这一点可能与其他面向对象程序设计语言略有区别。Python 中有许多内置对象可供编程者直接使用，例如数字、字符串、列表、元组、字典、集合、del 命令以及 cmp()、len()、id()、type()等大量内置函数；有些对象则需要导入模块（有些模块还需要单独进行安装）后才能使用，如 math 模块中的正弦函数 sin()，random 模块中的随机数产生函数 random()等。

在 Python 中内置了大量的常用对象，这些内置对象或类型无须导入模块即可直接使用，Python 常用内置对象如表 1-3 所示。

1.4.2 Python 变量

在 Python 中，不需要事先声明变量名及其类型，直接赋值即可创建各种类型的对象变量。例如语句

```
>>>x=3
```

创建了整型变量 x，并赋值为 3。

表 1-3　Python 常用内置对象

对象类型	示　　例	对象类型	示　　例
数字	1234，3.14，3+4j	文件	f=open('data.dat'，'r')
字符串	'swfu'，"I'm student"，"Python"	集合	set('abc')，{'a'，'b'，'c'}
列表	[1，2，3]，['a'，'b'，['c'，2]]	布尔型	True，False
字典	{1:'food'，2:'taste'，3:'import'}	空类型	None
元组	(2，−5，6)	编程单元类型	函数(使用 def 定义) 类(使用 class 定义)

需要说明的是，Python 属于强类型编程语言，虽然不需要在使用之前显式地声明变量及其类型，但是 Python 解释器会根据赋值或运算来自动推断变量类型。每种类型支持的运算也不完全一样，因此在使用变量时需要程序员自己确定所进行的运算是否合适，以免出现异常或者是意料之外的结果。后面大家会看到，同一个运算符对于不同类型的数据操作含义和计算结果也是不一样的。另外，Python 还是一种动态类型语言，也就是说，变量的类型是可以变化的。例如：

```
>>>x =3
>>>print(type(x))
<class 'int'>
>>>x = 'Hello world.'
>>>print(type(x))
<class 'str'>
>>>x = [1,2,3]
>>>print(type(x))
<class 'list'>
```

在上面的代码中，首先创建了整型变量 x，然后又分别创建了字符串和列表类型的变量 x。当创建了字符串类型的变量 x 之后，之前创建的整型变量 x 自动失效，创建列表对象 x 之后，之前创建的字符串变量 x 自动失效。可以将该模型理解为状态机，即修改其类型或删除之前，变量将一直保持上次的类型。

在大多数情况下，如果变量出现在赋值运算符或复合赋值运算符(例如+=、*=等)的左边，则表示创建变量或修改变量的值，否则表示引用该变量的值，这一点同样适用于使用下标来访问列表、字典等可变序列以及其他自定义对象中元素的情况。例如：

```
>>>x =3                    #创建整型变量
>>>print(x**2)
9
>>>x +=6                   #修改变量值
>>>print(x)                #读取变量值并输出显示
9
>>>x = [1,2,3]             #创建列表对象
```

```
>>>print(x)
[1, 2, 3]
>>>x[1] =5                #修改列表元素值
>>>print(x)               #输出显示整个列表
[1, 5, 3]
>>>print(x[2])            #输出显示列表指定元素
3
```

后面会提到,字符串和元组属于不可变序列,这意味着不能通过下标的方式来修改其中的元素值,例如,下面的代码试图修改元组中的元素值时会抛出异常。

```
>>>x = (1,2,3)
>>>print x
(1, 2, 3)
>>>x[1] =5
Traceback (most recent call last):
  File "<pyshell# 7>", line 1, in <module>
    x[1] =5
TypeError: 'tuple' object does not support item assignment
```

另外,在 Python 中,允许多个变量指向同一个值,例如:

```
>>>x =3
>>>id(x)
1786684560
>>>y =x
>>>id(y)
1786684560
```

然而,需要注意的是,继续上面的示例代码,当为其中一个变量修改值以后,其内存地址将会变化,但这并不影响另一个变量,例如,接着上面的代码再继续执行下面的代码:

```
>>>x +=6
>>>id(x)
1786684752
>>>y
3
>>>id(y)
1786684560
```

在这段代码中,id()函数用来返回变量所指值的内存地址。可以看出,在 Python 中修改变量值的操作,并不是修改了变量的值,而是修改了变量的指向。这是因为 Python 解释器首先读取变量 x 原来的值,然后将其加 6,并将结果存放于内存中,最后将变量 x 指向该结果的内存空间,理解这一点对于以后的编程非常重要。Python 内存管理模式如图 1-6 所示。

Python 采用的是基于值的内存管理方式,如果为不同变量赋值为相同值,则在内存中只有一份该值,多个变量指向同一块内存地址,例如:

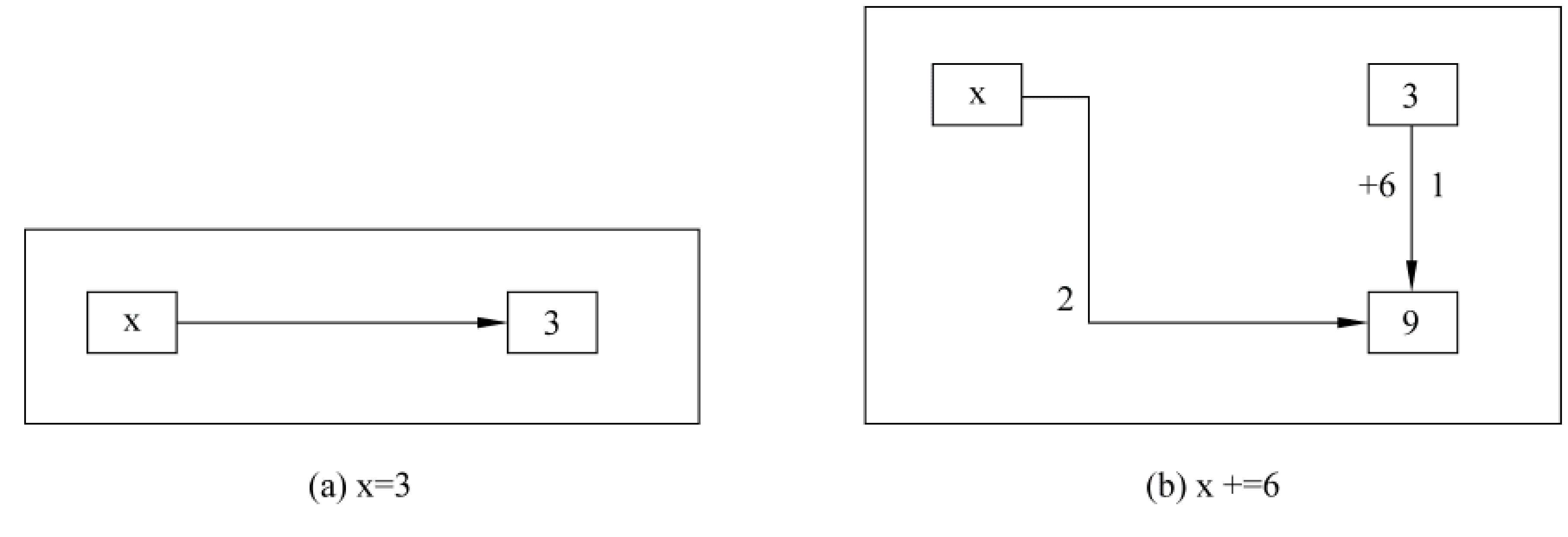


图 1-6　Python 内存管理模式

```
>>> x = 3
>>> id(x)
10417624
>>> y = 3
>>> id(y)
10417624
>>> y = 5
>>> id(y)
10417600
>>> id(x)
10417624
```

Python 具有自动内存管理功能，对于没有任何变量指向的值，Python 自动将其删除。Python 会跟踪所有的值，并自动删除不再有变量指向的值。因此，Python 程序员一般情况下不需要太多考虑内存管理的问题。尽管如此，显式释放不需要的值或显式关闭不再需要访问的资源，仍是一个好习惯，同时也是一个优秀的程序员的基本素养之一。

最后，在定义变量名的时候，需要注意以下问题。

（1）变量名必须以字母或下划线开头，但以下划线开头的变量在 Python 中有特殊含义，本书后面第 6 章会详细讲解。

（2）变量名中不能有空格和标点符号（括号、引号、逗号、斜线、反斜线、冒号、句号和问号等）。

（3）不能使用关键字作为变量名，可以导入 keyword 模块后使用 print（keyword. kwlist）查看所有 Python 关键字。

```
>>> import keyword
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else',
'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is',
'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while',
'with', 'yield']
>>> and = 3
SyntaxError: invalid syntax
```

（4）不建议使用系统内置的模块名、类型名或函数名作为变量名，这将会改变其类型和

含义，可以通过 dir(__builtins__)查看所有内置模块、类型和函数。

(5) 变量名对英文字母的大小写敏感，例如 student 和 Student 是不同的变量。

1.4.3 数字

数字属于 Python 不可变对象，1.4.2 节的例子已经说明了这一点，即修改整型变量值的时候并不是真的修改变量的值，而是先把值存放到内存中然后修改变量使其指向了新的内存地址，浮点数、复数等数字类型以及其他类型的变量具有同样的特点。列表、字典等可变类型对象的情况稍微复杂一些，将在第 2 章中进行详细讲解。

在 Python 中，数字类型变量可以表示任意大的数值，例如：

```
>>> a= 99999999999999999999999999999999
>>> a * a
9999999999999999999999999999999800000000000000000000000000000001L
>>> a * * 3
999999999999999999999999999999970000000000000000000000000000000299999999999999
999999999999999999L
```

当然，完全可以把 IDLE 当作计算器来使用，IDLE 可以实现复杂的数学运算，例如：

```
>>> 3 * (2+ 5)/3.0
7.0
>>> import math
>>> math.sqrt(3**2+ 4**2)
5.0
```

Python 数值类型主要有整数、浮点数和复数。整数类型主要有 4 种。

(1) 十进制整数。如 0、—1、9、123。

(2) 十六进制整数。使用 16 个数字 0、1、2、3、4、5、6、7、8、9、a、b、c、d、e、f 来表示整数，必须以 0x 开头，如 0x10、0xfa、0xabcdef。

(3) 八进制整数。使用 8 个数字 0、1、2、3、4、5、6、7 来表示整数，必须以 0o 开头，如 0o35、0o11。

(4) 二进制整数。使用 2 个数字 0、1 来表示整数，必须以 0b 开头，如 0b101、0b100。

浮点数也称为小数，例如.3、15.0、0.37、—11.2、1.2e2、314.15e—2，都是合法的浮点数。

Python 中的复数与数学上的复数形式一致，都是由实部和虚部构成，并且使用 j 或 J 来表示虚部。

```
>>> a = 3+ 4j
>>> b = 5+ 6j
>>> c = a+b
>>> c
(8+ 10j)
>>> c.real                #查看复数实部
8.0
>>> c.imag                #查看复数虚部
10.0
```

```
>>>a.conjugate()        #返回共轭复数
(3-4j)
>>>a*b
(-9+38j)
>>>a/b
(0.6393442622950819+0.03278688524590165j)
```

1.4.4 字符串

在 Python 中,字符串属于不可变序列,一般使用单引号、双引号或三引号进行界定,并且单引号、双引号、三单引号、三双引号还可以互相嵌套,用来表示复杂字符串。例如:

'abc'、'123'、'中国'、"Python"、'''Tom said,"Let's go"'''

都是合法字符串,空字符串表示为''或""或'''''',即一对不包含任何内容的任意字符串界定符。特别地,一对三单引号或三双引号表示的字符串支持换行,支持排版格式较为复杂的字符串,也可以在程序中表示较长的注释,后面将分别进行介绍。

由于字符串类型应用非常广泛,其支持的操作也较多,这里先简单介绍一下,后面第 4 章再结合正则表达式全面进行详细的讲解。

字符串支持使用+运算符进行合并以生成新字符串,例如:

```
>>>a = 'abc' + '123'
>>>a
'abc123'
```

可以对字符串进行格式化,把其他类型对象按格式要求转换为字符串,例如:

```
>>>a=3.6674
>>>'%7.3f' %a
'  3.667'
>>>"%d:%c"%(65,65)
'65:A'
>>>"""My name is %s, and my age is %d"""%('Dong Fuguo',38)
'My name is Dong Fuguo, and my age is 38'
```

Python 支持转义字符,常用的转义字符如表 1-4 所示。

表 1-4 转义字符

转义字符	含 义	转义字符	含 义
\n	换行符	\"	双引号
\t	制表符	\\	一个\
\r	回车	\ddd	3 位八进制数对应的字符
\'	单引号	\xhh	2 位十六进制数对应的字符

需要特别说明的是,字符串界定符前面加字母 r 或 R 表示原始字符串,其中的特殊字符不进行转义,但字符串的最后一个字符不能是\符号。原始字符串主要用于正则表达式,也

可以用来简化文件路径或 URL 的输入，请参考第 4 章的内容。

1.4.5 运算符与表达式

与其他语言一样，Python 支持大多数算术运算符、关系运算符、逻辑运算符以及位运算符。除此之外，还有一些运算符是 Python 特有的，例如成员测试运算符、集合运算符、同一性测试运算符等，Python 常用运算符如表 1-5 所示。

表 1-5 Python 常用运算符

<table>
<tr><th>运算符示例</th><th>功 能 说 明</th></tr>
<tr><td>x＋y</td><td>算术加法，列表、元组、字符串合并</td></tr>
<tr><td>x－y</td><td>算术减法，集合差集</td></tr>
<tr><td>x * y</td><td>乘法，序列重复</td></tr>
<tr><td>x/y</td><td>除法</td></tr>
<tr><td>x//y</td><td>求整商</td></tr>
<tr><td>－x</td><td>负数</td></tr>
<tr><td>x％y</td><td>余数/格式化(对实数可以进行余数运算)</td></tr>
<tr><td>x**y</td><td>幂运算</td></tr>
<tr><td>x＜y；x＜＝y；x＞y；x＞＝y</td><td>大小比较，集合的包含关系比较</td></tr>
<tr><td>x＝＝y；x！＝y</td><td>相等比较(值)，不等比较</td></tr>
<tr><td>x or y</td><td>逻辑或 (只有 x 为假才会计算 y)</td></tr>
<tr><td>x and y</td><td>逻辑与(只有 x 为真才会计算 y)</td></tr>
<tr><td>not x</td><td>逻辑非</td></tr>
<tr><td>x in y；x not in y</td><td>成员测试运算符</td></tr>
<tr><td>x is y；x is not y</td><td>对象实体同一性测试(地址)</td></tr>
<tr><td>|、^、&、<<、>>、~</td><td>位运算符</td></tr>
<tr><td>&、|</td><td>集合交集、并集</td></tr>
</table>

需要说明的是，Python 中的除法有两种，/和//分别表示除法和整除运算，并且 Python 2 和 Python 3 对这两种运算符的解释也略有区别，例如，在 Python 3.4.2 中运算结果如下：

```
>>>3/5
0.6
>>>3//5
0
>>>3.0/5
0.6
>>>3.0//5
0.0
```

而在 Python 2.7.8 中运算结果如下：

```
>>> 3/5
0
>>> 3//5
0
>>> 3.0/5
0.6
>>> 3.0//5
0.0
```

另外一个需要说明的，也是与其他有些语言略有不同的运算符是%。在 Python 中，除去前面已经介绍过的字符串格式化用法之外，该运算符还可以对整数和浮点数计算余数。但是由于浮点数的精确度的影响，计算结果可能略有误差，例如：

```
>>> 3.1%2
1.1
>>> 6.3%2.1
2.0999999999999996
>>> 6%2
0
>>> 6.0%2
0.0
>>> 6.0%2.0
0.0
>>> 5.7%4.8
0.9000000000000004
```

如前所述，Python 中很多运算符有多重含义，在程序中运算符的具体含义取决于操作数的类型。例如，* 运算符就是 Python 运算符中比较特殊的一个，它不仅可以用于数值乘法，还可以用于列表、字符串、元组等类型，当列表、字符串或元组等类型变量与整数进行 * 运算时，表示对内容进行重复并返回重复后的新对象。例如：

```
>>> 3 * 2                    #整数相乘
6
>>> 2.0 * 3                  #浮点数与整数相乘
6.0
>>> (3+4j) * 2               #复数与整数相乘
(6+8j)
>>> (3+4j) * (3-4j)          #复数与复数相乘
(25+0j)
>>> '1' * 5                  #字符串重复
'11111'
>>> "a" * 10                 #字符串重复
'aaaaaaaaaa'
>>> [1,2,3] * 3              #列表重复
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
>>> (1,2,3) * 3              #元组重复
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>>3* 'a'                    #字符串重复
'aaa'
```

在 Python 中，单个任何类型的对象或常数属于合法表达式，使用表 1-5 中运算符连接的变量和常量以及函数调用的任意组合也属于合法的表达式。例如：

```
>>>a = [1,2,3]
>>>b = [4,5,6]
>>>c =a+b
>>>c
[1, 2, 3, 4, 5, 6]
>>>d =map(str, c)
>>>d
['1', '2', '3', '4', '5', '6']
>>>import math
>>>map(math.sin, c)
[0.8414709848078965, 0.9092974268256817, 0.1411200080598672, -0.7568024953079282,
-0.9589242746631385, -0.27941549819892586]
>>>'Hello' +' ' +'world'
'Hello world'
>>>'welcome ' *  3
'welcome welcome welcome '
>>> ('welcome,' * 3).rstrip(',')+'!'
'welcome,welcome,welcome!'
```

1.4.6 常用内置函数

内置函数是指不需要导入任何模块即可直接使用的函数，例如，在 1.4.5 节中最后的例子中用到的 map()函数即属于 Python 内置函数，因此不需要导入任何模块就可以直接使用，该函数在本书后面会有讲解，当然读者也可以直接跳至第 5 章进行阅读，或者使用 help(map)来查看该函数帮助文档进行学习。

执行下面的命令可以列出所有内置函数：

```
>>>dir(__builtins__)
```

Python 常用的内置函数及其功能简要说明如表 1-6 所示。

表 1-6 Python 常用内置函数及其功能简要说明

函　　数	功能简要说明
abs(x)	返回数字 x 的绝对值
bin(x)	把数字 x 转换为二进制串
chr(x)	返回 ASCII 编码为 x 的字符

续表

函　　数	功能简要说明
dir()	返回指定对象的成员列表
eval(s[, globals[, locals]])	计算字符串中表达式的值并返回
float(x)	把数字或字符串 x 转换为浮点数并返回
help(obj)	返回对象 obj 的帮助信息
hex(x)	把数字 x 转换为十六进制串
id(obj)	返回对象 obj 的标识(地址)
input([提示内容字符串])	接受键盘输入,返回字符串。Python 2 和 Python 3 对该函数的解释不完全一样,详见后面的 1.4.8 节
len(obj)	返回对象 obj 包含的元素个数,适用于列表、元组、集合、字典和字符串等类型的对象
oct(x)	把数字 x 转换为八进制串
ord(s)	返回 1 个字符 s 的编码
range([start,] end [, step])	返回一个等差数列(Python 3 中返回一个 range 对象),不包括终值
round(x [, 小数位数])	对 x 进行四舍五入,若不指定小数位数,则返回整数
str(obj)	把对象 obj 转换为字符串
int(x[,d])	返回数字的整数部分,或把 d 进制的字符串 x 转换为十进制并返回,d 默认为十进制
list(x)、set([obj])、tuple(x)	把对象转换为列表、集合或元组并返回
max(x)、min(x)、sum(x)	返回序列中的最大值、最小值或数值元素之和
pow(x,y)	返回 x 的 y 次方
sorted(列表[,cmp[,key[reverse]]]	返回排序后的列表
type(obj)	返回对象 obj 的类型
reversed(列表或元组)	返回逆序后的列表或迭代器对象
map(函数,序列)	将单参数函数映射至序列中的每个元素,返回结果列表
reduce(函数,序列)	将接收 2 个参数的函数以累积的方式从左到右依次应用至序列中每个元素,最终返回单个值作为结果

由于内置函数众多且功能强大,很难一下子全部解释清楚,本书将根据内容组织的需要逐步展开演示其用法。这里只通过几个例子来演示部分内置函数的使用,如果读者需要用到某个内置函数而还没有看到本书后面的讲解,可以通过内置函数 help()查看函数的使用帮助,提前进行学习。

建议:编写程序时应优先考虑使用内置函数,因为内置函数不仅成熟、稳定,而且速度相对较快。

ord()和 chr()是一对功能相反的函数,ord()用来返回单个字符的序数或 ASCII 码,而 chr()则用来返回介于 0～255 之间的某序数对应的字符,str()则直接将其参数转换为字符

串。例如：

```
>>>ord('a')
97
>>>ord('A')
65
>>>chr(65)
'A'
>>>chr(67)
'C'
>>>chr(ord('A')+1)
'B'
>>>str(1)
'1'
>>>str(1234)
'1234'
>>>str([1,2,3])
'[1, 2, 3]'
>>>str((1,2,3))
'(1, 2, 3)'
>>>str({1,2,3})
'set([1, 2, 3])'
```

max()、min()、sum()这 3 个内置函数分别用于计算列表、元组或其他可迭代对象中所有元素最大值、最小值以及所有元素之和，sum()只支持数值型元素的序列或可迭代对象，max()和 min()则需要序列或可迭代对象中的元素之间可比较大小。例如，下面的示例代码，首先使用列表推导式生成包含 10 个随机数的列表，然后分别计算该列表的最大值、最小值和所有元素之和：

```
>>>import random
>>>a = [random.randint(1,100) for i in range(10)]
>>>a
[72, 26, 80, 65, 34, 86, 19, 74, 52, 40]
>>>print(max(a),min(a),sum(a))
86 19 548
```

很显然，如果需要计算该列表中的所有元素的平均值，可以使用下面的方法：

```
>>>a = [72, 26, 80, 65, 34, 86, 19, 74, 52, 40]
>>>sum(a) * 1.0/len(a)        #Python 2.7.8
54.8
```

对于初学者而言，也许 dir()和 help()这两个函数是最有用的，使用 dir()函数可以查看指定模块中包含的所有成员或者指定对象类型所支持的操作，而 help()函数则返回指定模块或函数的说明文档，这对于了解和学习新的模块与知识是非常重要的，能够熟练使用这两个函数也是学习能力的重要体现。

下面的代码首先导入数学模块 math，然后查看该模块的常量和函数，并查看指定函数

的使用帮助：

```
>>> import math
>>> dir(math)                #查看模块中的可用对象
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor',
'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
>>> dir(3+4j)                #查看数字类型成员
['__abs__', '__add__', '__class__', '__coerce__', '__delattr__', '__div__', '__divmod__',
'__doc__', '__eq__', '__float__', '__floordiv__', '__format__', '__ge__',
'__getattribute__', '__getnewargs__', '__gt__', '__hash__', '__init__', '__int__',
'__le__', '__long__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__',
'__nonzero__', '__pos__', '__pow__', '__radd__', '__rdiv__', '__rdivmod__',
'__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__', '__rmul__',
'__rpow__', '__rsub__', '__rtruediv__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__', '__truediv__', 'conjugate', 'imag', 'real']
>>> dir('')                  #查看字符串类型成员
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
'__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'_formatter_field_name_split', '_formatter_parser', 'capitalize', 'center',
'count', 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index',
'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
>>> help(math.sqrt)          #查看指定方法的使用帮助
Help on built-in function sqrt in module math:
sqrt(…)
    sqrt(x)
    Return the square root of x.
>>> help(math.sin)
Help on built-in function sin in module math:
sin(…)
    sin(x)
    Return the sine of x (measured in radians).
```

1.4.7 对象的删除

正如前面所提到的，在 Python 中具有自动内存管理功能，Python 解释器会跟踪所有的值，一旦发现某个值不再有任何变量指向，将会自动删除该值。尽管如此，自动内存管理或

者垃圾回收机制并不能保证及时释放内存。显式释放自己申请的资源是程序员的好习惯之一，也是程序员素养的重要体现之一。

在 Python 中，可以使用 del 命令来删除对象并解除与值之间的指向关系。删除对象时，如果其指向的值还有别的变量指向则不删除该值，如果删除对象后该值不再有其他变量指向，则删除该值。例如下面的代码所演示：

```
>>> x = [1,2,3,4,5,6]
>>> y = 3
>>> z = y
>>> print(y)
3
>>> del y              #删除对象
>>> print(y)
Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    print(y)
NameError: name 'y' is not defined
>>> print(z)
3
>>> del z
>>> print(z)
Traceback (most recent call last):
  File "<pyshell#56>", line 1, in <module>
    print(z)
NameError: name 'z' is not defined
>>> del x[1]           #删除列表中指定元素
>>> print(x)
[1, 3, 4, 5, 6]
>>> del x              #删除整个列表
>>> print(x)
Traceback (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

正如运行结果所示，变量 y 和 z 指向同一个值，删除变量 y 以后该值仍存在且被 z 所指向。另外，del 可以用来删除列表或其他可变序列中的指定元素，也可以删除整个列表或其他类型序列对象。列表中部分元素删除以后，列表会自动收缩其内存空间以保证各元素连续存储，这在后面第 2 章会详细介绍。del 无法删除元组或字符串中的指定元素，而只可以删除整个元组或字符串，因为这两者均属于不可变序列。

```
>>> x = (1,2,3)
>>> del x[1]
Traceback (most recent call last):
  File "<pyshell#62>", line 1, in <module>
```

```
        del x[1]
TypeError: 'tuple' object doesn't support item deletion
>>>del x
>>>print(x)
Traceback (most recent call last):
    File "<pyshell#64>", line 1, in <module>
        print(x)
NameError: name 'x' is not defined
```

1.4.8 基本输入输出

用 Python 进行程序设计，输入是通过 input()函数来实现的，input()的一般格式为

```
x=input('提示: ')
```

该函数返回用户输入的对象。

尽管形式一样，Python 2 和 Python 3 对该函数的解释略有不同。在 Python 2 中，该函数返回结果的类型由输入值时的界定符来决定，例如下面的 Python 2.7.8 代码：

```
>>>x =input("Please input:")
Please input:3                    #没有界定符，整数
>>>print type(x)
<type 'int'>
>>>x =input("Please input:")
Please input:'3'                  #单引号，字符串
>>>print type(x)
<type 'str'>
>>>x =input("Please input:")
Please input:[1,2,3]              #方括号，列表
>>>print type(x)
<type 'list'>
```

在 Python 2 中，还有一个 raw_input()函数用来接收用户输入的值，该函数返回结果的类型一律为字符串，而不管用户使用什么界定符。例如：

```
>>>x =raw_input("Please input:")
Please input:[1,2,3]
>>>print type(x)
<type 'str'>
```

在 Python 3 中，不存在 raw_input()函数，而 input()函数的返回结果是字符串，需要将其转换为相应的类型再处理，相当于 Python 2 中的 raw_input()函数。例如：

```
>>>x =input('Please input:')
Please input:3
>>>print(type(x))
<class 'str'>
>>>x =input('Please input:')
```

```
Please input:'1'
>>>print(type(x))
<class 'str'>
>>>x = input('Please input:')
Please input:[1,2,3]
>>>print(type(x))
<class 'str'>
>>>x = raw_input('Please input:')
Traceback (most recent call last):
    File "<pyshell#83>", line 1, in <module>
        x = raw_input('Please input:')
NameError: name 'raw_input' is not defined
```

Python 2 和 Python 3 的输出方法也不完全一致。在 Python 2 中，使用 print 语句进行输出，而在 Python 3 中使用 print()函数进行输出，上面的例子已经说明了这个区别。在本书第一篇后面章节中给出的代码中，大部分是在 Python 3.4.2 环境下编写的，而第二篇由于用到了一些暂时还不支持 Python 3 的扩展库，因此有些代码是使用 Python 2.7.8 编写的。当您偶尔遇到某个代码中使用 print 语句进行输出的话，我想您会明白原因的，并且也知道如何根据您安装的 Python 版本进行适当的改写。

默认情况下，Python 将结果输出到 IDLE 或者标准控制台，在输出时也可以进行重定向，例如可以把结果输出到指定文件。在 Python 2.7.8 中使用下面的方法进行输出重定向：

```
>>>fp = open(r'C:\mytest.txt','a+')
>>>print >>fp, "Hello,world"
>>>fp.close()
```

而在 Python 3.4.2 中则需要使用下面的方法进行重定向：

```
>>>fp = open(r'D:\mytest.txt', 'a+')
>>>print('Hello,world!', file = fp)
>>>fp.close()
```

另外一个重要的不同是，对于 Python 2 而言，在 print 语句之后加上逗号(,)则表示输出内容之后不换行，例如：

```
>>>for i in range(10):
    print i,
0 1 2 3 4 5 6 7 8 9
```

在 Python 3 中，为了实现上述功能则需要使用下面的方法：

```
>>>for i in range(10,20):
    print(i, end=' ')
10 11 12 13 14 15 16 17 18 19
```

在这两个示例中，range()是内置函数，用来生成一个列表或迭代对象，相信您已经明白该函数的基本用法，更加详细和巧妙的用法会在后面逐步展开。

1.4.9 模块

Python 默认安装仅包含部分基本或核心模块，但用户可以很方便地安装大量的扩展模块，pip 是管理扩展模块的重要工具。同样，在 Python 启动时，也仅加载了很少的一部分模块，在需要时由程序员显式地加载(有些模块可能需要先安装)其他模块。这样可以减小程序运行的压力，仅加载真正需要的模块和功能，且具有很强的可扩展性。可以使用 sys. modules. items()显示所有预加载模块的相关信息。

正如上面所述，对于很多模块而言，需要首先导入，然后才能使用其中的对象。Python 中主要有以下两种导入模块的方法。

1. import 模块名［as 别名］

使用这种方式导入模块以后，需要在要使用的对象之前加上前缀，即以“模块名. 对象名”的方式进行访问。也可以为导入的模块设置一个别名，然后使用“别名. 对象名”的方式来访问其中的对象。

```
>>> import math
>>>math.sin(0.5)              #求 0.5 的正弦值
0.479425538604203
>>> import random
>>>x =random.random()          #获得[0,1) 内的随机小数
>>>x
0.7866224717141462
>>>y =random.random(
>>>y
0.21054341257255382)
>>>n =random.randint(1,100)   #获得[1,100]上的随机整数
>>>n
82
>>>import numpy as np          #导入模块并设置别名
>>>a =np.array((1,2,3,4))     #通过模块的别名来访问其中的对象
>>>print a
[1 2 3 4]
```

2. from 模块名 import 对象名[as 别名]

使用这种方式仅导入明确指定的对象，并且可以为导入的对象起一个别名。这种导入方式可以减少查询次数，提高访问速度，同时也减少了程序员需要输入的代码量，而不需要使用模块名作为前缀。例如：

```
>>> from math import sin
>>> sin(3)
0.1411200080598672
>>> from math import sin as f
>>> f(3)
0.141120008059867
```

比较极端的情况是一次导入模块中的所有对象，例如：

```
from math import *
```

使用这种方式固然简单省事，但是并不推荐使用，一旦多个模块中有同名的对象，这种方式将会导致混乱。

有时候在测试自己编写的模块时，可能需要频繁地修改代码并重新导入模块，在 Python 2 中可以使用 reload()函数重新导入一个模块，而在 Python 3 中，需要使用 imp 模块的 reload()函数。

在导入模块时，Python 首先在当前目录中查找需要导入的模块文件，如果没有找到则从 sys 模块的 path 变量所指定的目录中查找，如果仍没有找到模块文件则提示模块不存在。可以使用 sys 模块的 path 变量查看 Python 导入模块时搜索模块的路径，也可以使用 append()方法向其中添加自定义的目录以扩展搜索路径。在导入模块时，会优先导入相应的.pyc 文件，如果相应的.pyc 文件与.py 文件时间不相符或不存在对应的.pyc 文件，则导入.py 文件并重新将该模块文件编译为.pyc 文件。关于 Python 文件名的详细介绍请参考 1.6 节的内容。

近年来，大量用于不同领域和专业的 Python 扩展库不断涌现，下面仅仅列出了其中很小的一部分。

(1) os：跨平台的操作系统模块，支持文件与文件夹以及其他相关操作。

(2) sys：提供了与解释器使用和维护有关对象的接口。

(3) math：提供了常用的数学函数。

(4) Locale：提供了 C 语言本地化函数的接口，并提供相关函数实现基于当前 locale 设置的数字与字符串转换。

(5) random：提供了随机数生成函数以及随机化有关的函数。

(6) pickle：支持序列化功能。

(7) datetime：支持日期时间有关功能。

(8) time：支持时间统计与测试有关功能。

(9) wmi：Windows 管理接口，需单独安装。

(10) tkinter：GUI 开发支持。

(11) urllib/urllib2：网页读取与访问。

(12) Pygame：游戏开发模块。

(13) wxPython：GUI 编程。

(14) SciPy：科学计算模块。

(15) PIL：图像处理模块。

(16) fabric：远程操作与部署。

(17) capstone：反汇编框架。

(18) ropper：ROP 相关框架。

(19) IDAPython：IDA 插件，使得 Python 程序可以运行于 IDA 中以实现可执行文件的自定义分析。

(20) Yara：恶意软件识别与分类引擎。

1.5 Python 代码编写规范

（1）缩进。

Python 程序依靠代码块的缩进来体现代码之间的逻辑关系。对于类定义、函数定义、选择结构、循环结构以及异常处理结构来说，行尾的冒号以及下一行的缩进表示一个代码块的开始，而缩进结束就表示一个代码块结束了。在编写程序时，同一个级别的代码块的缩进量必须相同。

在 IDLE 开发环境中，一般以 4 个空格为基本缩进单位，或者使用下面的方式来修改基本缩进量，如图 1-7 所示。

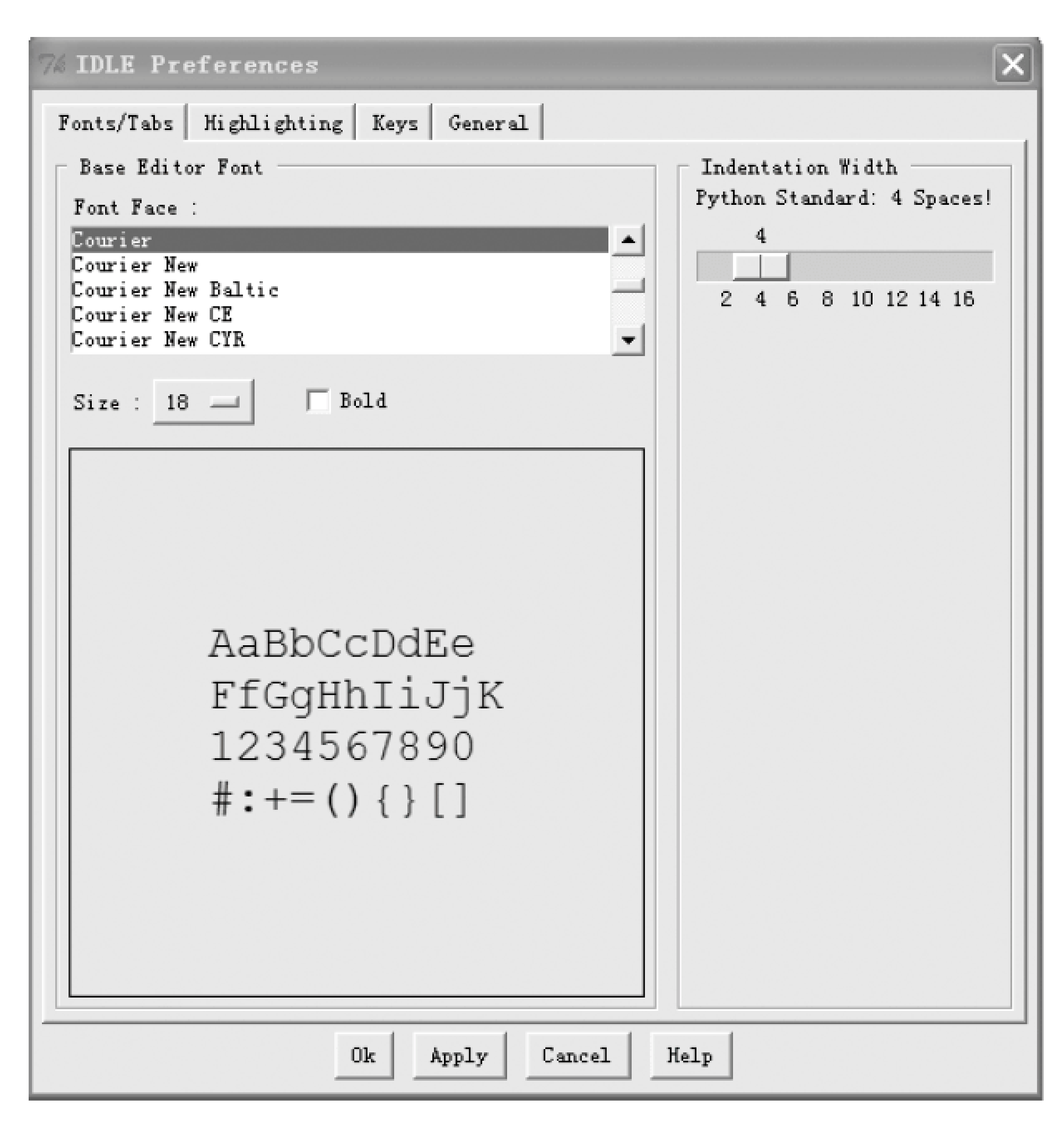


图 1-7　IDLE 环境中基本缩进量的设置

编写程序时，可以通过下面的菜单进行代码块的批量缩进和反缩进：

```
Fortmat→Indent Region/Dedent Region
```

当然，也可以使用快捷键 Ctrl+]进行缩进，使用快捷键 Ctrl+[进行反缩进。

（2）注释。

据统计，一个好的可维护性和可读性都很强的程序一般包含 30%以上的注释，注释对于程序理解和团队合作开发具有非常重要的意义。Python 中常用的注释方式主要有两种。

① 以#开始，表示本行#之后的内容为注释。

② 包含在一对三引号'''…'''或"""…"""之间且不属于任何语句的内容将被解释器认为是注释。

在 IDLE 开发环境中，可以通过下面的操作快速注释/解除注释代码块：

```
Format→Comment Out Region/Uncomment Region
```

或者也使用快捷键 Alt+3 和 Alt+4 进行代码块的批量注释和解除注释。

(3) 每个 import 语句只导入一个模块，尽量避免一次导入多个模块。

(4) 如果一行语句太长，可以在行尾使用续行符\来表示下面紧接的一行仍属于当前语句，但是一般建议使用括号来包含多行内容。

(5) 使用必要的空格与空行增强代码的可读性。运算符两侧、函数参数之间、逗号两侧建议使用空格进行分隔，而不同功能的代码块之间、不同的函数定义以及不同的类定义之间则建议增加一个空行以增加可读性。

(6) 适当使用异常处理结构提高程序容错性，但不能过多依赖异常处理结构。

(7) 软件应具有较强的可测试性，测试与开发齐头并进。

下面的代码可以用来检查 Python 程序的规范性，当然也可以在此基础上继续完善。

```
#-*-coding:utf-8-*-
#Filename: CheckCodeFormats.py
import sys,re

def checkFormats(lines, desFileName):

    fp = open(desFileName, 'w')

    for i, line in enumerate(lines):
        print '=' * 30
        print 'Line:', i+1

        if line.strip().startswith('#'):
            print ' ' * 10+'Pass.'
            fp.write(line)
            continue

        flag = True

        #check operator symbols
        symbols = [',', '+', '-', '*', '/', '//', '**', '>>', '<<', '+=', '-=',
        '*=', '/=']
        temp_line = line
        for symbol in symbols:
            pattern = re.compile(r'\s*'+re.escape(symbol)+r'\s*')
            temp_line = pattern.split(temp_line)
            sep = ' '+symbol+' '
            temp_line = sep.join(temp_line)
        if line != temp_line:
            flag = False
            print ' ' * 10+'You may miss some blank spaces in this line.'
            line = temp_line
```

```
            # check import statement
            if line.strip().startswith('import'):
                if ',' in line:
                    flag = False
                    print ' ' * 10+ "You'd better import one module at a time."
                    temp_line = line.strip()
                    modules = temp_line[temp_line.index(' ')+1:]
                    modules = modules.strip()
                    pattern = re.compile(r'\s * ,\s * ')
                    modules = pattern.split(modules)
                    temp_line = ''
                    for module in modules:
                        temp_line += line[:line.index('import')]+ 'import '+module+ '\n'
                    line = temp_line

                pri_line = lines[i- 1].strip()
                if pri_line and (not pri_line.startswith('import')) and \
                    (not pri_line.startswith('#')):
                    flag = False
                    print ' ' * 10+ 'You should add a blank line before this line.'
                    line = '\n'+ line

                after_line = lines[i+ 1].strip()
                if after_line and (not after_line.startswith('import')):
                    flag = False
                    print ' ' * 10+ 'You should add a blank line after this line.'
                    line = line+ '\n'

        # check if there is a blank line before new funtional code block, including the class/
        function definition
        if line.strip() and not line.startswith(' ') and i > 0:
            pri_line = lines[i- 1]
            if pri_line.strip() and pri_line.startswith(' '):
                flag = False
                print ' ' * 10+ "You'd better add a blank line before this line."
                line = '\n'+ line

        if flag:
            print ' ' * 10 + 'Pass.'
        fp.write(line)
    fp.close()

if __name__ == '__main__':
    fileName = 'CheckCodeFormats.py'# sys.argv[1]
```

```
fileLines = []
with open(fileName, 'r') as fp:
    fileLines = fp.readlines()
desFileName = fileName[:-3]+'_new.py'
checkFormats(fileLines, desFileName)

#check the ratio of comment lines to all lines
comments = [line for line in fileLines if line.strip().startswith('#')]
ratio = len(comments) * 1.0/len(fileLines)
if ratio <= 0.3:
    print '=' * 30
    print 'Comments in the file is less than 30%.'
    print 'Perhaps you should add some comments at appropriate position.'
```

1.6 Python 文件名

在 Python 中，不同扩展名的文件有不同的含义和用途，常见的扩展名主要有以下 5 种。

(1) py：Python 源文件，由 Python 解释器负责解释执行。

(2) pyw：Python 源文件，常用于图形界面程序文件。

(3) pyc：Python 字节码文件，无法使用文本编辑器正常查看文件内容。对于 Python 模块，第一次被导入时将被编译成字节码的形式，并在以后再次导入时优先使用 pyc 文件，以提高模块的加载和运行速度。对于非模块文件，直接执行时并不生成 pyc 文件，但可以使用 py_compile 模块的 compile()函数进行编译以提高加载和运行速度。

(4) pyo：优化的 Python 字节码文件，同样无法使用文本编辑器正常查看其内容。可以使用“python -O -m py-compile file.py”或“python -OO -m py-compile file.py”进行优化编译。

(5) pyd：一般是由其他语言编写并编译的二进制文件，常用于实现某些软件工具的 Python 编程接口或 Python 动态链接库。

1.7 Python 程序的运行方式

每个 Python 脚本在运行时都有一个__name__属性。如果脚本作为模块被导入，则其__name__属性的值被自动设置为模块名；如果脚本独立运行，则其__name__属性值被设置为'__main__'。

利用该属性即可控制 Python 程序的某些行为。例如，编写一个包含大量可被其他程序利用的函数的模块，而不希望该模块可以直接运行，则可以在程序文件中添加如下代码：

```
if __name__ == '__main__':
    print 'Please use me as a module.'
```

这样一来，程序直接执行时将会得到提示“Please use me as a module.”，而使用 import 语句将其作为模块导入后可以使用其中的函数或其他成员。

1.8　编写自己的包

包是 Python 用来组织命名空间和类的方式，可以看作是包含大量 Python 程序模块的文件夹。在包的每个目录中都必须包含一个__init__.py 文件，该文件可以是一个空文件，仅用于表示该目录是一个包。__init__.py 文件的主要用途是设置__all__变量以及执行初始化包所需的代码，其中__all__变量中定义的对象可以在使用 from …import * 时全部被正确导入。

假设有如下结构的包：

```
sound/                      Top-level package
    __init__.py             Initialize the sound package
    formats/                Subpackage for file format conversions
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ⋮
    effects/                Subpackage for sound effects
        __init__.py
        echo.py
        surround.py
        reverse.py
        ⋮
    filters/                Subpackage for filters
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ⋮
```

那么，就可以在自己的程序中使用下面的代码导入其中一个模块：

```
import sound.effects.echo
```

然后使用完整名字来访问或调用其中的成员，例如：

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

或者参考 1.4.9 节中介绍的使用模块成员的方法来访问该模块中的其他成员。

1.9　Python 快速入门

在了解前面的 Python 基础知识之后，让我们通过几个小程序来快速了解如何使用 Python 来解决实际的问题。就像前面介绍的一样，这几个例子的代码是以脚本程序的方式

给出的，所以需要在 IDLE 中创建一个程序文件，然后再输入这里的代码，最后保存为扩展名为.py 的文件并运行。

【例 1-1】 用户输入一个三位自然数，计算并输出其百位、十位和个位上的数字。

这个例子主要演示 Python 中算术运算符的用法，而计算每位上的数字有多种方法，这里只给出其中一种，您能再想出几种呢？哪一种方法的计算量最小呢？

```
x = input('请输入一个三位数：')
a = x//100
b = x//10%10
c = x%10
print(a, b, c)
```

注：与大部分程序设计教材一样，本书中给出的代码并不是完整的代码，只是为了演示特定的功能用法。例如，在本例中，完整的程序还要考虑用户输入是否为数字、是否为三位数等，可以使用 if…else…选择结构在计算之前进行判断，也可以使用异常处理结果来增加程序的健壮性和容错性，类似问题后面不再赘述。

【例 1-2】 已知三角形的两边长及其夹角，求第三边长。

这里需要用到 math 模块中求平方根的函数 sqrt()，当然这里给出的是比较传统的写法，参考前面的知识，相信您可以写出更加简洁的代码。

```
import math
x = input('输入两边长及夹角(度)：')
a, b, sita = x
c = math.sqrt(a * * 2 + b * * 2 - 2 * a * b * math.cos(sita * math.pi/180))
print 'c=',c
```

在这段代码中使用到了序列解包的知识，在后面会详细讲解，这里可以不必深究，用心体会 Python 的精妙和强大即可。

【例 1-3】 任意输入 3 个英文单词，按字典顺序输出。

在本例中，主要注意变量值交换的方法。

```
s = input('x,y,z=')
x, y, z = s.split(',')
if x > y:
    x, y = y, x
if x > z:
    x, z = z, x
if y > z:
    y, z = z, y
print(x, y, z)
```

1.10 Python 之禅

在本章的最后，让我们来用心读一下“Python 之禅”，笔者认为并不需要翻译这一段话，也不必进行过多的解读。只需要用心去体会，并在自己编写程序的时候多想想这段话，努力

让自己编写的代码更加优雅、更加 Pythonic。

```
Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea—let's do more of those!
```

本章知识精要

(1) 选择 Python 版本时应首先充分了解自己的需求和可用的扩展库情况。

(2) 应熟练掌握 Python 扩展库管理工具 pip。

(3) 在 Python 中一切都是对象。

(4) Python 采用的是基于值的内存管理方式。

(5) 编程时优先考虑使用内置函数来实现自己的业务逻辑。

(6) del 命令既可以删除一个变量,也可以删除列表等可变序列的部分元素。

(7) 可以使用 import 语句来导入模块中的对象。

(8) Python 程序使用缩进来体现代码之间的逻辑关系,并且建议使用必要的空格、空行和注释来提高程序的可读性。

习　题

1. 简单说明如何选择正确的 Python 版本。
2. 为什么说 Python 采用的是基于值的内存管理模式?
3. 在 Python 中导入模块中的对象有哪几种方式?
4. 使用 pip 命令安装 numpy 模块和 scipy 模块。
5. 编写程序,用户输入一个三位以上的整数,输出其百位以上的数字。例如,用户输入 1234,则程序输出 12(提示:使用整除运算)。

第 2 章　Python 数据结构

序列是程序设计中经常用到的数据存储方式，几乎每一种程序设计语言都提供了类似的数据结构，如 C 和 Basic 中的一维、多维数组等。简单地说，序列是一块用来存放多个值的连续内存空间，同一个序列中的元素通常是相关的，并且按一定顺序排列。Python 提供的序列类型可以说是所有程序设计语言的类似数据结构中最灵活的，也是功能最强大的。

Python 中常用的序列结构有列表、元组、字典、字符串、集合以及 range 对象等。除了字典和集合属于无序序列之外，列表、元组、字符串等序列均支持双向索引，第一个元素下标为 0，第二个元素下标为 1，以此类推；最后一个元素下标为−1，倒数第二个元素下标为−2，以此类推。以负数作为序列索引是 Python 语言的一大特色，熟练掌握和运用可以大幅度提高程序开发效率。

大量经验表明，熟练掌握 Python 基本数据结构（尤其是序列）可以更加快速有效地解决实际问题。本章通过大量案例介绍列表、元组、字典、集合等几种基本数据结构的用法，同时还有 range 函数的巧妙应用，以及在实际应用中非常有用的列表推导式。在本章的最后，介绍如何使用 Python 序列来实现栈、队列、树、图等较为复杂的数据结构。

2.1　列　　表

列表是 Python 的内置可变序列，是包含若干元素的有序连续内存空间，列表中的每一个数据称为元素，列表的所有元素放在一对中括号“([和])”中，并使用逗号分隔开。当列表增加或删除元素时，列表对象自动进行内存的扩展或收缩，从而保证元素之间没有缝隙，但这涉及列表元素的移动，效率较低，应尽量从列表尾部进行元素的增加与删除操作以提高处理速度。

注意： 优先考虑从列表尾部进行元素的增加与删除操作，会大幅度提高列表的处理速度。

在 Python 中，同一个列表中元素的数据类型可以各不相同，可以同时分别为整数、实数、字符串等基本类型，甚至是列表、元组、字典、集合以及其他自定义类型的对象。例如：

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
['spam', 2.0, 5, [10, 20]]
[['file1', 200,7], ['file2', 260,9]]
```

都是合法的列表对象。

对于 Python 序列而言，有很多方法是通用的，而不同类型的序列又有一些特有的方法。列表对象常用的方法如表 2-1 所示。除此之外，Python 的很多内置函数和命令也可以对列表和其他序列对象进行操作，后面将通过一些案例逐步进行介绍。

表 2-1 列表对象常用的方法

方　法	说　明
list. append(x)	将元素 x 添加至列表尾部
list. extend(L)	将列表 L 中的所有元素添加至列表尾部
list. insert(index, x)	在列表指定位置 index 处添加元素 x
list. remove(x)	在列表中删除首次出现的指定元素
list. pop([index])	删除并返回列表对象指定位置的元素，默认为最后一个元素
list. clear()	删除列表中所有元素，但保留列表对象，该方法在 Python 2 中没有
list. index(x)	返回值为 x 的首个元素的下标，若元素不存在则抛出异常
list. count(x)	返回指定元素 x 在列表中的出现次数
list. reverse()	对列表元素进行原地翻转
list. sort()	对列表元素进行原地排序
list. copy()	返回列表对象的浅复制，该方法在 Python 2 中没有

2.1.1 列表创建与删除

如同其他类型的 Python 对象变量一样，使用＝直接将一个列表赋值给变量即可创建列表对象，例如：

```
>>>a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
```

或者也可以使用 list()函数将元组、range 对象、字符串或其他类型的可迭代对象类型的数据转换为列表。例如：

```
>>>a_list = list((3,5,7,9,11))
>>>a_list
[3, 5, 7, 9, 11]
>>>list(range(1,10,2))
[1, 3, 5, 7, 9]
>>>list('hello world')
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
```

这里再次用到内置函数 range()，该函数语法为

```
range([start,] stop[, step])
```

该函数接受 3 个参数：第一个参数表示起始值（默认为 0）；第二个参数表示终止值（结果中不包括这个值）；第三个参数表示步长（默认为 1），函数返回一个 range 对象（在 Python 2 中返回一个包含整数的列表）。

当不再使用列表时，使用 del 命令删除整个列表，如果列表对象所指向的值不再有其他对象指向，Python 将同时删除该值。

```
>>>del a_list
```

```
>>> a_list
Traceback (most recent call last):
    File "<pyshell#6>", line 1, in <module>
        a_list
NameError: name 'a_list' is not defined
```

正如上面代码所展示的那样，删除列表对象 a_list 之后，该对象就不存在了，再次访问时将抛出异常 NameError，提示访问的对象名不存在。

2.1.2 列表元素的增加与删除

在实际应用中，列表元素的动态增加和删除也是经常遇到的操作，Python 列表提供了多种不同的方法来实现这一功能，本节对此做了比较详细的讲解和对比。

（1）可以使用+运算符来实现将元素添加到列表中的功能。虽然这种用法在形式上比较简单并且容易理解，但严格意义上来讲，这并不是真的为列表添加元素，而是创建一个新列表，并将原列表中的元素和新元素依次复制到新列表的内存空间。由于涉及大量元素的复制，该操作速度较慢，在涉及大量元素添加时不建议使用该方法。

```
>>> aList = [3,4,5]
>>> aList = aList + [7]
>>> aList
[3, 4, 5, 7]
```

（2）使用列表对象的 append()方法，原地修改列表，是真正意义上的在列表尾部添加元素，速度较快，也是推荐使用的方法。

```
>>> aList.append(9)
>>> aList
[3, 4, 5, 7, 9]
```

为了比较+和 append()这两种方法的速度差异，请看以下代码：

```
import time
result = []
start = time.time()
for i in range(10000):
    result = result + [i]
print(len(result), ',', time.time() - start)
result = []
start = time.time()
for i in range(10000):
    result.append(i)
print(len(result), ',', time.time() - start)
```

在上面代码中，分别重复执行 10 000 次+运算和 append()方法为列表插入元素，使用 time 模块的 time()函数返回当前时间，然后运行代码之后计算时间差。其中一次的运行结果如下，可以看出，这两个方法的速度相差还是非常大的，使用 append()方法比使用+运算

快约 70 倍。

```
10000, 0.21801209449768066
10000, 0.003000020980834961
```

前面曾经提到过，Python 采用的是基于值的自动内存管理方式，当为对象修改值时，并不是真的直接修改变量的值，而是使变量指向新的值，这对于 Python 所有类型的变量都是一样的，例如下面的代码：

```
>>> a = [1,2,3]
>>> id(a)
20230752
>>> a = [1,2]
>>> id(a)
20338208
```

然而，对于列表、集合、字典等可变序列类型而言，情况稍微复杂一些。以列表为例，列表中包含的是元素值的内存地址，而不是直接包含元素值。如果是直接修改序列变量的值，则与 Python 普通变量的情况是一样的，而如果是通过下标来修改序列中元素的值或通过可变序列对象自身提供的方法来增加和删除元素时，序列对象在内存中的起始地址是不变的。例如下面的代码：

```
>>> a= [1,2,4]
>>> b= [1,2,3]
>>> a==b
False
>>> id(a)==id(b)
False
>>> id(a[0])==id(b[0])
True
>>> a = [1,2,3]
>>> id(a)
25289752
>>> a.append(4)
>>> id(a)
25289752
>>> a.remove(3)
>>> a
[1, 2, 4]
>>> id(a)
25289752
>>> a[0] =5
>>> a
[5, 2, 4]
>>> id(a)
25289752
```

```
>>>a.extend([7,8,9])
>>>a
[5, 2, 4, 7, 8, 9]
>>>id(a)
25289752
```

(3) 使用列表对象的 extend()方法可以将另一个迭代对象的所有元素添加至该列表对象尾部。正如上面的代码所示,通过 extend()方法来增加列表元素也不改变其内存首地址,属于原地操作。

```
>>>aList.extend([11,13])
>>>aList
[3, 4, 5, 7, 9, 11, 13]
>>>aList.extend((15,17))
>>>aList
[3, 4, 5, 7, 9, 11, 13, 15, 17]
```

(4) 使用列表对象的 insert()方法将元素添加至列表的指定位置。

```
>>>aList.insert(3,6)
>>>aList
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
```

正如本节开始所说,应尽量从列表尾部进行元素的增加与删除操作。列表的 insert()方法可以在列表的任意位置插入元素,但由于列表的自动内存管理功能,insert()方法会涉及插入位置之后所有元素的移动,这会影响处理速度,类似的还有后面介绍的 remove()方法。因此,除非有必要,否则应尽量避免在列表中间位置插入和删除元素的操作,而是优先考虑使用前面介绍的 append()方法。下面的代码演示了 insert()方法和 append()方法的效率。

```
import time
def Insert():
    a = []
    for i in range(10000):
        a.insert(0,i)

def Append():
    a = []
    for i in range(10000):
        a.append(i)
start = time.time()
for i in range(10):
    Insert()
print 'Insert:',time.time()-start
start = time.time()
for i in range(10):
    Append()
```

```
print 'Append:',time.time()-start
```

运行结果如下，可以看到这两个方法的速度有很大差异，后面的列表元素删除方法也具有同样的特点，不再赘述。

```
Insert: 0.578000068665
Append: 0.0309998989105
```

(5) 使用乘法来扩展列表对象，将列表与整数相乘，生成一个新列表，新列表是原列表中元素的重复。

```
>>>aList = [3,5,7]
>>>bList =aList
>>>id(aList)
57091464
>>>id(bList)
57091464
>>>aList =aList * 3
>>>aList
[3, 5, 7, 3, 5, 7, 3, 5, 7]
>>>bList
[3,5,7]
>>>id(aList)
57092680
>>>id(bList)
57091464
```

通过上面代码的运行结果可以看出，该操作实际上是创建了一个新列表，而不是扩展原列表。该操作同样适用于字符串和元组。

(6) 使用 del 命令删除列表中指定位置上的元素。前面已经提到过，del 命令也可以直接删除整个列表，这里不再赘述。

```
>>>del a_list[1]
>>>a_list
[3, 7, 9, 11]
```

(7) 使用列表的 pop()方法删除并返回指定(默认为最后一个)位置上的元素，如果给定的索引超出了列表的范围则抛出异常。

```
>>>a_list =list((3,5,7,9,11))
>>>a_list.pop()
11
>>>a_list
[3, 5, 7, 9]
>>>a_list.pop(1)
5
>>>a_list
[3, 7, 9]
```

（8）使用列表对象的 remove()方法删除首次出现的指定元素，如果列表中不存在要删除的元素，则抛出异常。

```
>>>a_list = [3,5,7,9,7,11]
>>>a_list.remove(7)
>>>a_list
[3, 5, 9, 7, 11]
```

有时可能需要删除列表中指定元素的所有重复，大家会很自然地想到使用“循环＋remove()”的方法，但是具体操作时很有可能会出现意料之外的错误，代码运行没有出现错误，但结果是错的。例如，下面的代码试图删除列表中所有的 1，使用下面的代码从前向后遍历列表元素，遇到 1 则删除，然而当循环结束后却发现只删除了一半，还有一半并没有删除。

```
>>>a_list = [1,1,1,1,1,1,1,1,1,1,1,1]
>>>len(a_list)
12
>>>for i in a_list:
    if i ==1:
        a_list.remove(i)
>>>a_list
[1, 1, 1, 1, 1, 1]
>>>len(a_list)
6
```

出现这个问题的原因是列表的自动内存管理功能，前面已经提到，在删除列表元素时，Python 会自动对列表内存进行收缩以保证所有元素之间没有空隙，这样的话，每当删除一个元素之后，该元素后面所有元素的索引就都改变了。为了避免这样的问题，可以使用下面代码演示的方法来删除列表中指定元素的所有重复。

```
>>>a_list = [1,1,1,1,1,1,1,1,1,1,1,1]
>>>for i in a_list[::-1]:
    if i ==1:
        a_list.remove(i)
>>>a_list
[]
>>>len(a_list)
0
```

2.1.3　列表元素访问与计数

如同其他语言里的数组一样，可以使用下标直接访问列表中的元素。如果指定下标不存在，则抛出异常提示下标越界，例如：

```
>>>aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>>aList[3]
```

```
6
>>>aList[3] =5.5
>>>aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>>aList[15]
Traceback (most recent call last):
    File "<pyshell#34>", line 1, in <module>
        aList[15]
IndexError: list index out of range
```

使用列表对象的 index()方法可以获取指定元素首次出现的下标，语法为 index(value, [start, [stop]])，其中 start 和 stop 用来指定搜索范围，start 默认为 0，stop 默认为列表长度。若列表对象中不存在指定元素，则抛出异常提示列表中不存在该值，例如：

```
>>>aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>>aList.index(7)
4
>>>aList.index(100)
Traceback (most recent call last):
    File "<pyshell#36>", line 1, in <module>
        aList.index(100)
ValueError: 100 is not in list
```

如果需要知道指定元素在列表中出现的次数，可以使用列表对象的 count()方法进行统计，例如：

```
>>>aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>>aList.count(7)
1
>>>aList.count(0)
0
```

2.1.4　成员资格判断

如果需要判断列表中是否存在指定的值，可以使用前面介绍的 count()方法，如果存在则返回大于 0 的数，如果返回 0 则表示不存在。或者，使用更加简洁的 in 关键字来判断一个值是否存在于列表中，返回结果为 True 或 False。

```
>>>aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>>3 in aList
True
>>>18 in aList
False
>>>bList = [[1],[2],[3]]
```

```
>>> 3 in bList
False
>>> 3 not in bList
True
>>> [3] in bList
True
>>> aList = [3, 5, 7, 9, 11]
>>> bList = ['a', 'b', 'c', 'd']
>>> (3, 'a') in zip(aList, bList)
True
>>> for a,b in zip(aList,bList):
    print a,b
3 a
5 b
7 c
9 d
```

关键字 in 和 not in 也可以用于其他可迭代对象，包括元组、字典、range 对象、字符串和集合等，常用在循环语句中对序列或其他可迭代对象中的元素进行遍历，在前面的例子中已经见过这个用法，后面还会多次用到。使用这种方法来遍历序列或迭代对象，可以减少代码的输入量、简化程序员的工作，并且大幅度提高程序的可读性，推荐熟练掌握和运用。

2.1.5 切片操作

切片是 Python 序列的一个重要操作，适用于列表、元组、字符串、range 对象等类型。切片使用 2 个冒号分隔的 3 个数字来完成，第一个数字表示切片的开始位置(默认为 0)，第二个数字表示切片截止(但不包含)位置(默认为列表长度)，第三个数字表示切片的步长(默认为 1)，当步长省略时可以顺便省略最后一个冒号。可以使用切片来截取列表中的任何部分，得到一个新列表，也可以通过切片来修改和删除列表中的部分元素，甚至可以通过切片操作为列表对象增加元素。

与前面介绍的使用下标访问元素的方法不同，切片操作不会因为下标越界而抛出异常，而是简单地返回一个空列表或在列表尾部截断。

```
>>> aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::]
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::-1]
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> aList[::2]
[3, 5, 7, 11, 15]
>>> aList[1::2]
[4, 6, 9, 13, 17]
>>> aList[3::]
[6, 7, 9, 11, 13, 15, 17]
>>> aList[3:6]
```

```
[6, 7, 9]
>>> aList[3:6:1]
[6, 7, 9]
>>> aList[0:100:1]
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> a[100:]
[]
```

可以使用切片操作来快速实现很多目的，如原地修改列表内容，列表元素的增、删、改、查以及元素替换等操作都可以通过切片来实现，并且不影响列表对象内存地址。

```
>>> aList = [3,5,7]
>>> aList[len(aList):]
[]
>>> aList[len(aList):] = [9]
>>> aList
[3, 5, 7, 9]
>>> aList[:3] = [1,2,3]
>>> aList
[1, 2, 3, 9]
>>> aList[:3] = []
>>> aList
[9]
>>> aList = list(range(10))
>>> aList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> aList[::2] = [0] * (len(aList)//2)
>>> aList
[0, 1, 0, 3, 0, 5, 0, 7, 0, 9]
```

也可以结合使用 del 命令与切片操作来删除列表中的部分元素。

```
>>> aList = [3,5,7,9,11]
>>> del aList[:3]
>>> aList
[9, 11]
```

需要注意的是，切片返回的是列表元素的浅复制，与列表对象的赋值并不一样。例如下面的代码：

```
>>> aList = [3,5,7]
>>> bList = aList          #bList 与 aList 指向同一个内存
>>> bList
[3, 5, 7]
>>> bList[1] = 8
>>> aList
[3, 8, 7]
>>> aList == bList
```

```
True
>>> aList is bList
True
>>> id(aList)            #这里的输出结果很可能和您的不一样,这是正常的
19061816
>>> aList= [3,5,7]
>>> id(bList)
19061816
>>> bList = aList[::]    #浅复制
>>> aList == bList
True
>>> aList is bList
False
>>> id(aList) == id(bList)
False
>>> bList[1] = 8
>>> bList
[3, 8, 7]
>>> aList
[3, 5, 7]
>>> aList == bList
False
>>> aList is bList
False
>>> id(aList)
19061816
>>> id(bList)
11656168
>>> cmp(aList,bList)     #内置函数 cmp()的知识请参考 2.1.7 节
-1
```

2.1.6 列表排序

在实际应用中,经常需要对列表元素进行排序,一个很自然的想法就是使用列表对象自身提供的 sort()方法进行原地排序,该方法支持多种不同的排序方式。

```
>>> aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> import random
>>> random.shuffle(aList)          #打乱顺序
>>> aList
[3, 4, 15, 11, 9, 17, 13, 6, 7, 5]
>>> aList.sort()                   #默认为升序排列
>>> aList
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList.sort(reverse = True)     #也可以降序排列
>>> aList
```

```
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>>aList.sort(key=lambda x:len(str(x)))      #自定义排序
>>>aList
[9, 7, 6, 5, 4, 3, 17, 15, 13, 11]
```

也可以使用内置函数 sorted()对列表进行排序，与列表对象的 sort()方法不同，该函数返回新列表，并不对原列表进行任何修改。

```
>>>aList
[9, 7, 6, 5, 4, 3, 17, 15, 13, 11]
>>>sorted(aList)
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>>sorted(aList,reverse=True)
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
```

在某些应用中可能需要将列表元素进行逆序排列，也就是所有元素位置反转，第一个变成最后一个，第二个变成倒数第二个，以此类推。可以使用列表对象的 reverse()方法将元素原地逆序：

```
>>>import random
>>>aList=[random.randint(50,100) for i in range(10)]
>>>aList
[87, 79, 52, 96, 56, 59, 74, 80, 53, 79]
>>>aList.reverse()
>>>aList
[79, 53, 80, 74, 59, 56, 96, 52, 79, 87]
```

Python 提供了内置函数 reversed()支持对列表元素进行逆序排列，与列表对象的 reverse()方法不同，该函数不对原列表做任何修改，而是返回一个逆序排列后的迭代对象。例如：

```
>>>aList=[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>>newList=reversed(aList)
>>>newList
<listreverseiterator object at 0x0000000003624198>
>>>list(newList)
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>>for i in newList:
    print i,
```

在上面代码中，最后的 for 循环没有输出任何内容，因为在之前的 list()函数执行时，迭代对象已遍历结束，需要重新创建迭代对象才能再次访问其内容，即：

```
>>>newList=reversed(aList)
>>>for i in newList:
    print i,
17 15 13 11 9 7 6 5 4 3
```

2.1.7 用于序列操作的常用内置函数

由于序列的重要性和广泛应用，Python 提供了大量可用于序列操作的内置函数，在 1.4.6 节已经介绍了几个，本节再通过示例来简单扩展一下。

(1) cmp(列表 1,列表 2)：对两个列表进行比较，若第一个列表大于第二个，则结果为 1，否则为－1，元素完全相同则结果为 0，类似于＝＝运算符，但和 is、is not 不一样。例如：

```
>>> (1, 2, 3) < (1, 2, 4)
True
>>> cmp((1, 2, 3) , (1, 2, 4))
-1
>>> [1, 2, 3] < [1, 2, 4]
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> (1, 2, 3, 4) < (1, 2, 4)
True
>>> (1, 2) < (1, 2, -1)
True
>>> cmp((1, 2), (1, 2, -1))
-1
>>> (1, 2, 3) == (1.0, 2.0, 3.0)
True
>>> cmp((1, 2, 3), (1.0, 2.0, 3.0))
0
>>> (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
True
>>> cmp('a','A')
1
```

在 Python 3.x 中不支持 cmp()函数，可以使用关系运算符比较序列大小。

(2) len(列表)：返回列表中的元素个数，同样适用于元组、字典、集合、字符串和 range 对象等各种可迭代对象。

(3) max(列表)、min(列表)：返回列表中的最大或最小元素，同样适用于元组、字符串、集合、range 对象和字典等。但对字典进行操作时，默认是对字典的“键”进行计算，如果需要对字典的“值”进行计算，则需要使用字典对象的 values()方法明确说明。

```
>>> a={1:1,2:5,3:8}
>>> max(a)
3
>>> max(a.values())
8
```

(4) sum(列表)：对数值型列表的元素进行求和运算，对非数值型列表运算则出错，同样适用于元组、集合、range 对象、字典等。但对字典进行操作时，默认是对字典“键”进行计

算，如果需要对字典“值”进行计算，则需要使用字典对象的 values()方法明确说明。

```
>>>a={1:1,2:5,3:8}
>>>sum(a)
6
>>>sum(a.values())
14
```

(5) zip(列表 1,列表 2,…)：将多个列表或元组对应位置的元素组合为元组，并返回包含这些元组的列表(Python 2. x)或 zip 对象(Python 3. x)。例如，在 Python 2. 7. 8 中代码运行如下：

```
>>>aList = [1,2,3]
>>>bList = [4,5,6]
>>>cList = [7,8,9]
>>>dList = zip(aList, bList, cList)
>>>dList
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

而在 Python 3. 4. 2 中则需要这样使用：

```
>>>aList = [1,2,3]
>>>bList = [4,5,6]
>>>cList=zip(a,b)
>>>cList
<zip object at 0x0000000003728908>
>>>list(cList)
[(1, 4), (2, 5), (3, 6)]
```

(6) enumerate(列表)：枚举列表、元组或其他可迭代对象的元素，返回枚举对象，枚举对象中每个元素是包含下标和元素值的元组。该函数对字符串、字典同样有效。但对字典进行操作时，默认是对字典“键”进行计算，如果需要对字典“值”进行计算，则需要使用字典对象的 values()方法明确说明。

```
>>>for item in enumerate(cList):
    print item
(0, (1, 4))
(1, (2, 5))
(2, (3, 6))
>>>for index, ch in enumerate('SDIBT'):
    print((index,ch),end=',')
(0, 'S'),(1, 'D'),(2, 'I'),(3, 'B'),(4, 'T'),
>>>a
{1: 1, 2: 5, 3: 8}
>>>for i,v in enumerate(a):
    print i,v
0 1
```

```
1 2
2 3
>>> for i,v in enumerate(a.values()):
    print i,v
0 1
1 5
2 8
```

2.1.8 列表推导式

列表推导式可以说是 Python 程序开发时应用最多的技术之一。前面曾经使用列表推导式来快速生成包含多个随机数的列表，可以看出，列表推导式使用非常简洁的方式来快速生成满足特定需求的列表，代码具有非常强的可读性。例如：

```
>>> aList = [x * x for x in range(10)]
```

相当于：

```
>>> aList = []
>>> for x in range(10):
    aList.append(x * x)
```

而

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> aList = [w.strip() for w in freshfruit]
```

则等价于下面的代码：

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> for i,v in enumerate(freshfruit):
    freshfruit[i] =v.strip()
```

同时，也等价于

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> freshfruit = list(map(str.strip,freshfruit))
```

但是不等价于下面的代码：

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> for i in freshfruit:
    i =i.strip()
```

下面通过几个示例来进一步体会列表推导式的强大功能。

（1）使用列表推导式实现嵌套列表的平铺。

```
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

(2) 过滤不符合条件的元素。

在列表推导式中可以使用 if 子句来进行筛选，例如，下面的代码可以列出当前文件夹下的所有 Python 源文件：

```
>>> import os
>>> [filename for filename in os.listdir('.') if filename.endswith('.py')]
```

下面的代码用于从列表中选择符合条件的元素组成新的列表：

```
>>> aList = [-1,-4,6,7.5,-2.3,9,-11]
>>> [i for i in aList if i>0]
[6, 7.5, 9]
```

再如，已知有一个包含一些同学成绩的字典，计算成绩的最高分、最低分和平均分，并查找所有最高分的同学，代码可以这样编写：

```
>>> scores = {"Zhang San": 45, "Li Si": 78, "Wang Wu": 40, "Zhou Liu": 96, "Zhao Qi": 65, "Sun Ba": 90, "Zheng Jiu": 78, "Wu Shi": 99, "Dong Shiyi": 60}
>>> highest = max(scores.values())
>>> lowest = min(scores.values())
>>> highest
99
>>> lowest
40
>>> average = sum(scores.values()) * 1.0/len(scores)
>>> average
72.33333333333333
>>> highestPerson = [name for name, score in scores.items() if score == highest]
>>> highestPerson
['Wu Shi']
```

(3) 在列表推导式中使用多个循环，实现多序列元素的任意组合，并且可以结合条件语句过滤特定元素：

```
>>> [(x,y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

(4) 使用列表推导式实现矩阵转置：

```
>>> matrix = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> [[row[i] for row in matrix] for i in (4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

也可以使用内置函数 zip()和 list()来实现矩阵转置：

```
>>> list(zip(* matrix))
```

```
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

2.2 元　　组

与列表类似,元组也是 Python 的一个重要序列结构,但与列表不同的是,元组属于不可变序列。元组一旦创建,用任何方法都不可以修改其元素的值,也无法为元组增加或删除元素,如果确实需要修改的话,只能再创建一个新的元组。

元组的定义形式和列表相似,区别在于定义元组时所有元素放在一对圆括号中,即()中,而不是方括号中。

2.2.1 元组的创建与删除

使用=将一个元组赋值给变量,就可以创建一个元组变量。

```
>>>a_tuple = ('a', )
>>>a_tuple
('a')
>>>a_tuple = ('a', 'b', 'mpilgrim', 'z', 'example')
>>>a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
```

如果要创建只包含一个元素的元组,只把元素放在圆括号里是不行的,还需要在元素后面加一个逗号“,”,而创建包含多个元素的元组则不存在这个问题。

```
>>>a=3
>>>a
3
>>>a= (3)
>>>a
3
>>>a=3,
>>>a
(3,)
>>>a=1,2
>>>a
(1, 2)
```

如同使用 list()函数将序列转换为列表一样,也可以使用 tuple()函数将其他类型序列转换为元组。

```
>>>print tuple('abcdefg')
('a', 'b', 'c', 'd', 'e', 'f', 'g')
>>>aList
[-1, -4, 6, 7.5, -2.3, 9, -11]
>>>tuple(aList)
(-1, -4, 6, 7.5, -2.3, 9, -11)
```

对于元组而言，只能使用 del 命令删除整个元组对象，而不能只删除元组中的部分元素，因为元组属于不可变序列。

2.2.2 元组与列表的区别

列表属于可变序列，可以随意地修改列表中的元素值以及增加和删除列表元素，而元组属于不可变序列，元组中的数据一旦定义就不允许通过任何方式进行更改。因此，元组没有提供 append()、extend()和 insert()等方法，无法向元组中添加元素；同样，元组也没有 remove()和 pop()方法，也不支持对元组元素进行 del 操作，不能从元组中删除元素，只能使用 del 命令删除整个元组。元组也支持切片操作，但是只能通过切片来访问元组中的元素，而不支持使用切片来修改元组中元素的值，也不支持使用切片操作来为元组增加或删除元素。

Python 内置函数 tuple()可以接受一个列表、字符串或其他序列类型和迭代器作为参数，并返回一个包含同样元素的元组，而 list()函数可以接受一个元组、字符串或其他序列类型和迭代器作为参数并返回一个列表。从效果上看，tuple()函数可以看作是在冻结列表并使其不可变，而 list()是在融化元组使其可变。

元组的访问和处理速度比列表更快。如果定义了一系列常量值，主要用途仅是对它们进行遍历或其他类似用途，而不需要对其元素进行任何修改，那么一般建议使用元组而不用列表。可以认为元组对不需要修改的数据进行了“写保护”，从内在实现上不允许修改其元素值，从而使得代码更加安全。

最后，作为不可变序列，与整数、字符串一样，元组可用作字典的键，而列表则永远都不能当作字典键使用，因为列表是可变的。

2.2.3 序列解包

在实际开发中，序列解包是非常重要和常用的一个用法，可以以非常简洁的形式完成复杂的功能，大幅度提高了代码的可读性，并且减少了程序员的代码输入量。例如，可以使用序列解包功能对多个变量同时进行赋值：

```
>>>x,y,z =1,2,3
>>>x,y,z
(1, 2, 3)
>>>print x,y,z
1 2 3
```

再如：

```
v_tuple = (False, 3.5, 'exp')
>>> (x, y, z) =v_tuple
```

或者

```
>>>x, y, z =v_tuple
```

序列解包也可以用于列表和字典，但是对字典使用时，默认是对字典“键”进行操作，如果需要对键-值对进行操作，需要使用字典的 items()方法；如果需要对字典“值”进行操作，

则需要使用字典的 values()方法明确指定。对字典操作时,不需要对元素的顺序考虑过多。例如,下面的代码演示了列表与字典的序列解包操作:

```
>>>a=[1,2,3]
>>>b,c,d=a
>>>s={'a':1,'b':2,'c':3}
>>>b,c,d=s.items()
>>>b
('c', 3)
>>>b,c,d=s
>>>b
'c'
>>>b,c,d=s.values()
>>>print b,c,d
1 3 2
```

使用序列解包可以很方便地同时遍历多个序列。

```
>>>keys=['a','b','c','d']
>>>values=[1,2,3,4]
>>>for k,v in zip(keys,values):
    print k,v
a 1
b 2
c 3
d 4
```

在前面章节中关于内置函数 enumerate()的示例中,也是采用了序列解包的操作。再如,下面对字典的操作也使用到序列解包:

```
>>>s={'a':1,'b':2,'c':3}
>>>for k,v in s.items():
    print k,v

a 1
c 3
b 2
```

在调用函数时,在实参前面加上一个星号(*)也可以进行序列解包,从而实现将序列中的元素值依次传递给相同数量的形参,详见 5.3.4 节。

2.2.4 生成器推导式

从形式上看,生成器推导式与列表推导式非常接近,只是生成器推导式使用圆括号而不是列表推导式所使用的方括号。与列表推导式不同的是,生成器推导式的结果是一个生成器对象,而不是列表,也不是元组。使用生成器对象的元素时,需要将其转化为列表或元组,也可以使用生成器对象的 next()方法(Python 2.x)或__next__()方法(Python 3.x)进行遍

历，或者直接将其作为迭代器来使用。但是不管用哪种方法访问其元素，当所有元素访问结束以后，如果需要重新访问，必须重新创建该生成器对象。

```
>>>g=((i+2)**2 for i in range(10))
>>>g
<generator object <genexpr> at 0x02B15C60>
>>>tuple(g)                          #转化为元组
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
>>>tuple(g)                          #元素已经遍历结束
()
>>>g=((i+2)**2 for i in range(10))   #重新创建生成器对象
>>>list(g)                           #转化为列表
[4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>>g=((i+2)**2 for i in range(10))
>>>g.next()                          #单步迭代，在 Python 3 中应改为__next__()
4
>>>g.next()
9
>>>g.next()
16
>>>g.next()
25
>>>g=((i+2)**2 for i in range(10))
>>>for i in g:                       #直接进行循环迭代
    print i,
4 9 16 25 36 49 64 81 100 121
```

2.3 字　　典

字典是键-值对的无序可变序列，字典中的每个元素包含两部分：键和值。定义字典时，每个元素的键和值用冒号分隔，相邻元素之间用逗号分隔，所有的元素放在一对大括号中，即{}中。

字典中的键可以为任意不可变数据，比如整数、实数、复数、字符串和元组等，但不能使用列表、集合、字典作为字典的键，因为这些类型的对象是可变的。另外，字典中的键不允许重复，值是可以重复的。

可以使用内置函数 globals()返回和查看包含当前作用域内所有全局变量和值的字典，使用内置函数 locals()返回包含当前作用域内所有局部变量和值的字典。

```
>>>a = (1, 2, 3, 4, 5)
>>>b = 'Hello world.'
>>>def demo():
    a = 3
    b = [1,2,3]
    print 'locals:',locals()
```

```
    print 'globals:',globals()
>>> demo()
locals: {'a': 3, 'b': [1, 2, 3]}
globals: {'a': (1, 2, 3, 4, 5), 'b': 'Hello world.', '__builtins__': <module
'__builtin__' (built-in)>, 'demo': <function demo at 0x013907F0>, '__package__':
None, '__name__': '__main__', '__doc__': None}
```

2.3.1 字典创建与删除

与列表和元组的创建一样，使用＝将一个字典赋值给一个变量即可创建一个字典变量。

```
>>> a_dict = {'server': 'db.diveintopython3.org', 'database': 'mysql'}
>>> a_dict
{'database': 'mysql', 'server': 'db.diveintopython3.org'}
```

可以使用内置函数 dict()通过已有数据快速创建字典：

```
>>> keys= ['a','b','c','d']
>>> values= [1,2,3,4]
>>> dictionary= dict(zip(keys,values))
>>> print dictionary
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

或者使用内置函数 dict()根据给定的键-值对来创建字典：

```
>>> d= dict(name= 'Dong',age= 37)
>>> d
{'age': 37, 'name': 'Dong'}
```

还可以以给定内容为键，创建值为空的字典：

```
>>> adict= dict.fromkeys(['name','age','sex'])
>>> adict
{'age': None, 'name': None, 'sex': None}
```

不再需要某个字典时，可以使用 del 命令删除整个字典，也可以使用 del 命令删除字典中指定的元素，请参考后面的内容。

2.3.2 字典元素的读取

与列表和元组类似，可以使用下标的方式来访问字典中的元素，但不同的是字典的下标是字典的“键”，而列表和元组访问时下标必须为整数值。使用下标的方式访问字典值时，若指定的“键”不存在则抛出异常。

```
>>> aDict= {'name':'Dong', 'sex':'male', 'age':37}
>>> aDict['name']
'Dong'
>>> aDict['tel']
Traceback (most recent call last):
```

```
    File "<pyshell#53>", line 1, in <module>
        aDict['tel']
KeyError: 'tel'
```

目前推荐使用的且更加安全的字典元素访问方式是字典对象的 get()方法。使用字典对象的 get()方法可以获取指定键对应的值,并且可以在指定“键”不存在的时候返回指定值,如果不指定则默认返回 None。

```
>>>print(aDict.get('address'))
None
>>>print(aDict.get('address', 'SDIBT'))
SDIBT
>>>aDict['score'] =aDict.get('score',[])
>>>aDict['score'].append(98)
>>>aDict['score'].append(97)
>>>aDict
{'age': 37, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
```

另外,使用字典对象的 items()方法可以返回字典的键-值对列表,使用字典对象的 keys()方法可以返回字典的“键”列表,使用字典对象的 values()方法可以返回字典的“值”列表。

```
>>>aDict={'name':'Dong', 'sex':'male', 'age':37}
>>>for item in aDict.items():
    print item
('age', 37)
('name', 'Dong')
('sex', 'male')
>>>for key in aDict:
    print key
age
name
sex
>>>for key, value in aDict.items():
    print key, value
age 37
name Dong
sex male
>>>print aDict.keys()
['age', 'name', 'sex']
>>>aDict.values()
[37, 'Dong', 'male']
```

2.3.3 字典元素的操作

当以指定“键”为下标为字典元素赋值时,若该“键”存在,则表示修改该“键”的值;若该键不存在,则表示添加一个新的键-值对,也就是添加一个新元素。

```
>>> aDict['age'] = 38
>>> aDict
{'age': 38, 'name': 'Dong', 'sex': 'male'}
>>> aDict['address'] = 'SDIBT'
>>> aDict
{'age': 38, 'address': 'SDIBT', 'name': 'Dong', 'sex': 'male'}
```

使用字典对象的 update()方法将另一个字典的键-值对一次性全部添加到当前字典对象。

```
>>> aDict
{'age': 37, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> aDict.items()
[('age', 37), ('score', [98, 97]), ('name', 'Dong'), ('sex', 'male')]
>>> aDict.update({'a':'a','b':'b'})
>>> aDict
{'a': 'a', 'score': [98, 97], 'name': 'Dong', 'age': 37, 'b': 'b', 'sex': 'male'}
```

需要删除字典元素时，可以根据具体要求使用 del 命令删除字典中指定“键”对应的元素，或者也可以使用字典对象的 clear()方法来删除字典中的所有元素，还可以使用字典对象的 pop()方法删除并返回指定键的元素，或者使用字典对象的 popitem()方法删除并返回字典中的一个元素，大家可以自行练习这些用法。

2.4 集　合

集合是无序可变集合，与字典一样使用一对大括号作为界定符，同一个集合的元素之间不允许重复，集合中每个元素都是唯一的。

2.4.1 集合的创建与删除

正如前面多次提到的那样，在 Python 中变量不需要提前声明其类型，直接将集合赋值给变量即可创建一个集合对象。

```
>>> a = {3,5}
>>> a.add(7)
>>> a
set([3, 5, 7])
```

也可以使用 set()函数将列表、元组等其他可迭代对象转换为集合，如果原来的数据中存在重复元素，则在转换为集合时只保留一个。

```
>>> a_set=set(range(8,14))
>>> a_set
set([8, 9, 10, 11, 12, 13])
>>> b_set=set([0,1,2,3,0,1,2,3,7,8])
>>> b_set
```

```
set([0, 1, 2, 3, 7, 8])
```

不再使用某个集合时，可以使用 del 命令删除整个集合。另外，也可以使用集合对象的 pop()方法弹出并删除其中一个元素，或者使用集合对象的 remove()方法直接删除指定元素，以及使用集合对象的 clear()方法清空集合并删除所有元素。

```
>>>a={1,4,2,3}
>>>a.pop()
1
>>>a
set([2, 3, 4])
>>>a.pop()
2
>>>a
set([3, 4])
>>>a.add(2)
>>>a
set([2, 3, 4])
>>>a.remove(3)                          #删除指定元素
>>>a
set([2, 4])
>>>a.pop(2)                             #pop()方法不接受参数
Traceback (most recent call last):
    File "<pyshell#76>", line 1, in <module>
        a.pop(2)
TypeError: pop() takes no arguments (1 given)
```

2.4.2 集合操作

Python 集合支持交集、并集和差集等运算，读者结合在其他课程中学过的集合知识，应该不难理解下面的代码。

```
>>>a_set
set([8, 9, 10, 11, 12, 13])
>>>b_set
set([0, 1, 2, 3, 7, 8])
>>>a_set.union(b_set)                   #并集
set([0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13])
>>>a_set&b_set                          #交集
set([8])
>>>a_set.intersection(b_set)            #交集
set([8])
>>>a_set.difference(b_set)              #差集
set([9, 10, 11, 12, 13])
>>>a_set.symmetric_difference(b_set)    #对称差
set([0, 1, 2, 3, 7, 9, 10, 11, 12, 13])
```

```
>>>a_set^b_set
set([0, 1, 2, 3, 7, 9, 10, 11, 12, 13])
```

作为集合的具体应用,可以使用集合快速提取序列中的单一元素,即提取出序列中所有不重复的元素,如果使用传统方式的话,需要编写下面的代码:

```
>>>import random
>>>listRandom = [random.choice(range(10000)) for i in range(100)]
>>>noRepeat = []
>>>for i in listRandom :
    if i not in noRepeat :
        noRepeat.append(i)
>>>len(listRandom)
>>>len(noRepeat)
```

如果使用集合的话,只需要如下一行代码就可以了,可以参考上面的代码对结果进行验证。

```
>>>newSet = set(listRandom)
```

2.5 其他数据结构

在应用开发中,除了 Python 基本序列之外,还经常需要使用到其他一些数据结构,例如堆、栈、队列、树和图等。其中有些结构 Python 本身已经提供了,而有些则需要自己利用 Python 基本序列来实现。本节内容可以看作是 Python 序列、元组等基本数据结构的扩展,或者 Python 基本数据结构的二次开发。这里假设您对数据结构的知识有所了解,因此有些基本概念就不做过多的解释了,如果需要的话,可自行查阅有关资料。当然,也可以参考本节的思路自己编写代码来实现数据结构课程中更加复杂的数据结构。

2.5.1 堆

堆是一种重要的数据结构,在进行排序时使用较多,Python 在 heapq 模块中提供了对堆的支持。下面的代码演示了堆的用法,同时也请读者注意 random 模块的用法。

```
>>>import heapq
>>>import random
>>>data = range(10)
>>>data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>>random.choice(data)                    #随机选择
9
>>>random.choice(data)
1
>>>random.shuffle(data)                   #乱序
>>>data
[6, 1, 3, 4, 9, 0, 5, 2, 8, 7]
```

```
>>> heap = []
>>> for n in data:                    #建堆
    heapq.heappush(heap,n)
>>> heap
[0, 2, 1, 4, 7, 3, 5, 6, 8, 9]
>>> heapq.heappush(heap,0.5)
>>> heap
[0, 0.5, 1, 4, 2, 3, 5, 6, 8, 9, 7]
>>> heapq.heappop(heap)               #弹出最小的元素
0
>>> heapq.heappop(heap)
0.5
>>> heapq.heappop(heap)
1
>>> myheap = [1,2,3,5,7,8,9,4,10,333]
>>> heapq.heapify(myheap)
>>> myheap
[1, 2, 3, 4, 7, 8, 9, 5, 10, 333]
>>> heapq.heapreplace(myheap,6)
1
>>> myheap
[2, 4, 3, 5, 7, 8, 9, 6, 10, 333]
```

2.5.2 队列

队列的特点是"先进先出(FIFO)"和"后进后出(LILO)",在某些应用中有着重要的作用,例如多线程编程、作业处理等。Python 提供了 Queue 模块(在 Python 3 中为 queue)和 collections.deque 模块支持队列的操作,当然也可以使用 Python 列表进行二次开发来实现自定义的队列结构。例如,下面的代码演示了 Queue 模块的用法。

```
>>> import Queue                      # queue in Python3
>>> q=Queue.Queue()
>>> q.put(0)
>>> q.put(1)
>>> q.put(2)
>>> print q.queue
deque([0, 1, 2])
>>> print q.get()
0
>>> print q.queue
deque([1, 2])
>>> print q.get()
1
>>> print q.queue
deque([2])
```

下面的代码使用了 collections 模块的双端队列:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")
>>> queue.append("Graham")
>>> queue.popleft()
'Eric'
>>> queue.popleft()
'John'
>>> queue
deque(['Michael', 'Terry', 'Graham'])
```

下面的类利用 Python 列表实现了自定义的队列结构：

```
class myQueue:
    def __init__(self, size = 10):
        self._content = []
        self._size = size
    def setSize(self, size):
        self._size = size
    def put(self, v):
        if len(self._content) < self._size:
            self._content.append(v)
        else:
            print 'The queue is full'
    def get(self):
        if self._content:
            return self._content.pop(0)
        else:
            print 'The queue is empty'
    def show(self):
        if self._content:
            print self._content
        else:
            print 'The queue is empty'
    def empty(self):
        self._content = []
    def isEmpty(self):
        if not self._content:
            return True
        else:
            return False
    def isFull(self):
        if len(self._content) == self._size:
            return True
        else:
            return False
```

可以使用该类的方法来实现队列的基本操作，也可以进行扩展以实现其他特殊需求。下面的代码简单演示了这个自定义队列结构的用法：

```
>>> import myQueue
>>> q = myQueue.myQueue()
>>> q.get()
The queue is empty
>>> q.put(5)
>>> q.show()
[5]
>>> q.put(7)
>>> q.put('a')
>>> q.show()
[5, 7, 'a']
>>> q.isEmpty()
False
>>> q.isFull()
False
>>> q.get()
5
>>> q.get()
7
>>> q.get()
'a'
>>> q.get()
The queue is empty
```

2.5.3 栈

栈是一种"后进先出(LIFO)"或"先进后出(FILO)"的数据结构，Python 列表本身就可以实现栈结构的基本操作。列表对象的 append()方法是在列表尾部追加元素，类似于入栈操作；pop()方法默认是弹出并返回列表的最后一个元素，类似于出栈操作。但是直接使用 Python 列表对象模拟栈操作并不太方便，例如当列表为空时再执行 pop()则会抛出一个不很友好的异常；另外，也无法限制列表对象的大小。

```
>>> myStack = []
>>> myStack.append(3)
>>> myStack.append(5)
>>> myStack.append(7)
>>> myStack
[3, 5, 7]
>>> myStack.pop()
7
>>> myStack.pop()
5
>>> myStack.pop()
```

```
3
>>> myStack.pop()
Traceback (most recent call last):
    File "<pyshell#85>", line 1, in <module>
        myStack.pop()
IndexError: pop from empty list
```

下面的代码使用 Python 列表实现了自定义的栈结构来模拟栈的操作：

```
class Stack:
    def __init__(self, size = 10):
        self._content = []
        self._size = size
    def empty(self):
        self._content = []
    def isEmpty(self):
        if not self._content:
            return True
        else:
            return False
    def setSize(self, size):
        self._size = size
    def isFull(self):
        if len(self._content) == self._size:
            return True
        else:
            return False
    def push(self, v):
        if len(self._content) < self._size:
            self._content.insert(0, v)
        else:
            print 'Stack Full!'
    def pop(self):
        if len(self._content) > 0:
            v = self._content[0]
            del self._content[0]
            return v
        else:
            print 'Stack is empty!'
    def show(self):
        print self._content
```

将上面代码保存为 Stack.py 文件之后，可以作为模块进行使用来模拟栈的基本操作。当然，也可以在上面代码的基础上进行扩展以实现您的想法：

```
>>> import Stack
>>> s = Stack.Stack()
```

```
>>> s.push(3)
>>> s.show()
[3]
>>> s.push(5)
>>> s.show()
[5, 3]
>>> s.push(7)
>>> s.show()
[7, 5, 3]
>>> s.pop()
7
>>> s.show()
[5, 3]
```

2.5.4 链表

可以直接使用 Python 列表及其基本操作来实现链表的功能，当然也可以对列表进行封装来实现自定义的链表结构。下面的代码使用 Python 列表模拟了链表及其基本操作：

```
>>> linkTable = []
>>> linkTable.append(3)                  #在尾部追加节点
>>> linkTable.append(5)
>>> linkTable
[3, 5]
>>> linkTable.insert(1,4)                #在链表中间插入节点
>>> linkTable
[3, 4, 5]
>>> linkTable.remove(linkTable[1])       #删除节点
>>> linkTable
[3, 5]
```

2.5.5 二叉树

如果学过数据结构的话，大家肯定还对当时的痛苦记忆犹新，用 C/C++ 来实现二叉树要考虑很多问题，看到下面使用 Python 实现的二叉树，相信您会眼前一亮。

```
class BinaryTree:
    def __init__(self, value):
        self.__left = None
        self.__right = None
        self.__data = value
    def insertLeftChild(self, value):
        if self.__left:
            print '__left child tree already exists.'
        else:
            self.__left = BinaryTree(value)
```

```
            return self.__left
    def insertRightChild(self, value):
        if self.__right:
            print 'Right child tree already exists.'
        else:
            self.__right = BinaryTree(value)
            return self.__right
    def show(self):
        print self.__data
    def preOrder(self):
        print self.__data
        if self.__left:
            self.__left.preOrder()
        if self.__right:
            self.__right.preOrder()
    def postOrder(self):
        if self.__left:
            self.__left.postOrder()
        if self.__right:
            self.__right.postOrder()
        print self.__data
    def inOrder(self):
        if self.__left:
            self.__left.inOrder()
        print self.__data
        if self.__right:
            self.__right.inOrder()
if __name__ == '__main__':
    print 'Please use me as a module.'
```

可以看出，这段代码是把 print 作为输出语句来使用的，可以不进行任何修改地运行于 Python 2，如果您用的是 Python 3，仅需要把 print 语句改为 print() 函数即可，前面有过类似的说明，这里就不再赘述了。假设把上面的代码保存为文件 BinaryTree.py，然后可以使用下面的方法来使用上面定义的二叉树类。要注意的是，需要把这个文件放在 Python 的安装目录中，或者把含有该文件的目录添加到 sys.path 中。

```
>>> import BinaryTree
>>> root = BinaryTree.BinaryTree('root')
>>> firstLeft = root.insertLeftChild('A')
>>> firstRight = root.insertRightChild('B')
>>> secondLeft = firstLeft.insertLeftChild('C')
>>> thridRight = secondLeft.insertRightChild('D')
>>> root.postOrder()
D
C
```

```
A
B
root
>>> root.inOrder()
C
D
A
root
B
```

2.5.6 有向图

作为本章的最后一个示例，让我们来看一下使用 Python 语言实现有向图的创建和路径搜索。有向图由节点和边组成，而每条边都是有方向的，两个节点之间存在有向边则表示可以从起点到达终点。与二叉树的示例一样，这里给出稍微完整的代码。

```
#-*-coding:utf-8-*-
#Filename: DirectedGraph.py
#--------------------
#Function description:path searching of directed graph
#--------------------
def searchPath(graph, start, end):
    results = []
    __generatePath(graph, [start], end, results)
    results.sort(key=lambda x:len(x))
    return results
def __generatePath(graph, path, end, results):
    current=path[-1]
    if current ==end:
        results.append(path)
    else:
        for n in graph[current]:
            if n not in path:
                #path.append(n)
                __generatePath(graph, path + [n], end, results)
def showPath(results):
    print 'The path from ',results[0][0], ' to ', results[0][-1], ' is:'
    for path in results:
        print path
if __name__=='__main__':
    graph={'A':['B', 'C', 'D'],
           'B':['E'],
           'C':['D', 'F'],
           'D':['B', 'E', 'G'],
           'E':['G'],
```

```
            'F':['D', 'G'],
            'G':['E', 'A', 'B']}
r = searchPath(graph, 'A', 'D')
showPath(r)
```

为了节省篇幅，您可以自行运行该程序，并结合运行结果来理解这段代码。

本章知识精要

(1) 列表、字符串、元组都支持双向索引。

(2) 列表、字典和集合属于可变序列，元组和字符串属于不可变序列。

(3) 如果有可能，应尽量从列表的尾部进行元素的增加与删除操作。

(4) 切片操作不仅可以用来返回列表、元组、字符串中的部分元素，还可以对列表中的元素值进行修改，以及增加或删除列表中的元素。

(5) 列表推导式可以使用简洁的形式来生成满足特定需要的列表。

(6) 序列解包在多个场合具有重要的应用，是 Python 的基本操作之一。

(7) 字典中的元素是键-值对，其中的“键”不允许重复。

(8) 集合中的所有元素不允许重复。

习　题

1. 为什么应尽量从列表的尾部进行元素的增加与删除操作？

2. 编写程序，生成包含 1000 个 0～100 之间的随机整数，并统计每个元素的出现次数。

3. 编写程序，用户输入一个列表和两个整数作为下标，然后输出列表中介于两个下标之间的元素组成的子列表。例如，用户输入[1,2,3,4,5,6]和 2,5，程序输出[3,4,5,6]。

4. 设计一个字典，并编写程序，用户输入内容作为键，然后输出字典中对应的值，如果用户输入的键不存在，则输出“您输入的键不存在！”。

5. 编写程序，生成包含 20 个随机数的列表，然后将前 10 个元素升序排列，后 10 个元素降序排列，并输出结果。

6. 在 Python 中，字典和集合都是用一对________作为界定符，字典的每个元素有两部分组成，即________和________，其中________不允许重复。

7. 假设有列表 a=['name','age','sex']和 b=['Dong',38,'Male']，请使用一个语句将这两个列表的内容转换为字典，并且以列表 a 中的元素为键，以列表 b 中的元素为值，这个语句可以写为________。

8. 假设有一个列表 a，现要求从列表 a 中每 3 个元素取 1 个，并且将取到的元素组成新的列表 b，可以使用语句________。

9. 使用列表推导式生成包含 10 个数字 5 的列表，语句可以写为________。

10. ________(可以、不可以)使用 del 命令来删除元组中的部分元素。

第 3 章　选择与循环

在传统的面向过程程序设计中有 3 种经典的控制结构，即顺序结构、选择结构和循环结构。即使在面向对象程序设计语言中以及事件驱动或消息驱动应用开发中，也无法脱离这 3 种基本的程序结构。可以说，不管使用哪种程序设计语言，在实际开发中，为了实现特定的业务逻辑或算法，都不可避免地要用到大量的选择结构和循环结构，并且经常需要将选择结构和循环结构嵌套使用。在本章中，首先介绍 Python 中选择结构与循环结构的语法，然后通过几个示例来理解其用法。

3.1　运算符与条件表达式

在选择结构和循环结构中，都要使用条件表达式来确定下一步的执行流程。在条件表达式中可以使用 1.4.5 节介绍的如下所有运算符。

(1) 算术运算符：+、-、*、/、//、%、**。

(2) 关系运算符：>、<、==、<=、>=、!=。

(3) 测试运算符：in、not in、is、is not。

(4) 逻辑运算符：and、or、not。

(5) 位运算符：~、&、|、^、<<、>>。

在选择和循环结构中，条件表达式的值只要不是 False、0(或 0.0、0j 等)、空值 None、空列表、空元组、空集合、空字典、空字符串或其他空序列，Python 解释器均认为与 True 等价。从这个意义上来讲，几乎所有的 Python 合法表达式都可以作为条件表达式。例如：

```
>>>if 3:
    print(5)
5
>>>a = [1,2,3]
>>>if a:
    print(a)
[1, 2, 3]
>>>a = []
>>>if a:
    print a
else:
    print 'empty'

empty
>>>i =s =0
>>>while i <=10:
    s +=i
```

```
    i +=1
>>>print s
55
>>>i =s =0
>>>while True:
    s +=i
    i +=1
    if i >10:
        break
>>>print s
55
>>>s =0
>>>for i in range(0,11,1):
    s +=i
>>>print s
55
```

关于表达式和运算符的详细内容在 1.4.5 节中已有介绍，这里不再赘述，只简单介绍一下条件表达式中比较特殊的几个运算符。首先是关系运算符，与很多语言不同的是，在 Python 中的关系运算符可以连续使用，例如：

```
>>>print 1<2<3
True
>>>print 1<2>3
False
>>>print 1<3>2
True
```

比较特殊的运算符还有逻辑运算符 and 和 or，这两个运算符具有短路求值或惰性求值的特点，简单地说，就是只计算必须计算的表达式的值。在设计条件表达式时，在遇到复杂条件时如果能够巧妙利用逻辑运算符 and 和 or 的短路求值或惰性求值特性，可以大幅度提高程序的运行效率，减少不必要的计算与判断。以 and 为例，对于表达式“表达式 1 and 表达式 2”而言，如果“表达式 1”的值为 False 或其他等价值时，不论“表达式 2”的值是什么，整个表达式的值都是 False，此时“表达式 2”的值无论是什么都不影响整个表达式的值，因此将不会被计算，从而减少不必要的计算和判断。逻辑或运算符 or 也具有类似的特点，读者可以自行分析。在设计条件表达式时，如果能够大概预测不同条件失败的概率，并将多个条件根据 and 和 or 运算的短路求值特性进行排序，可以大幅度提高程序的运行效率。例如，下面的函数用来使用用户指定的分隔符将多个字符串连接成一个字符串，如果用户没有指定分隔符则使用逗号。

```
>>>def Join(chList, sep=None):
    return (sep or ',').join(chList)
>>>chTest =['1','2','3','4','5']
>>>Join(chTest)
'1,2,3,4,5'
```

```
>>>Join(chTest,':')
'1:2:3:4:5'
>>>Join(chTest,' ')
'1 2 3 4 5'
```

当然，还可以把上面的函数直接定义为下面的形式：

```
>>>def Join(chList, sep=','):
    return sep.join(chList)
```

3.2 选择结构

选择结构通过判断某些特定条件是否满足来决定下一步的执行流程，是非常重要的控制结构。常见选择结构的有单分支选择结构、双分支选择结构、多分支选择结构、嵌套的分支结构，形式比较灵活多变，具体使用哪一种最终取决于要实现的业务逻辑。从某种意义上讲，后面章节中讲到的循环结构和异常处理结构中也可以带有 else 子句，可以看作是选择结构的一种变形。

3.2.1 单分支选择结构

单分支选择结构是最简单的一种形式，其语法如下所示，其中的冒号是不可缺少的，表示一个语句块的开始。

```
if 表达式:
    语句块
```

当表达式值为 True 或其他等价值时，表示条件满足，语句块将被执行，否则该语句块将不被执行。

```
a,b=input('Input two numbers:')
if a>b:
    a,b=b,a
print a,b
```

3.2.2 双分支选择结构

双分支选择结构的语法为

```
if 表达式:
    语句块 1
else:
    语句块 2
```

当表达式的值为 True 或其他等价值时，执行语句块 1，否则执行语句块 2。该结构类似于下面的表达式语法：

```
value1 if condition else value2
```

例如：

```
>>>a =5
>>>print(6) if a>3 else print(5)
6
>>>print(6 if a>3 else 5)
6
>>>b =6 if a>13 else 9
>>>b
9
>>>chTest = ['1','2','3','4','5']
>>>if chTest:
    print chTest
else:
    print 'Empty'
['1', '2', '3', '4', '5']
```

3.2.3 多分选择支结构

多分支选择结构为用户提供了更多的选择，可以实现复杂的业务逻辑，多分支选择结构的语法为

```
if 表达式 1:
    语句块 1
elif 表达式 2:
    语句块 2
elif 表达式 3:
    语句块 3
⋮
else:
    语句块 n
```

其中，elif 是 else if 的缩写。下面的代码演示了利用多分支选择结构将成绩从百分制变换到等级制的用法。

```
>>>def func(score):
    if score >100:
        return 'wrong score.must <=100.'
    elif score >=90:
        return 'A'
    elif score >=80:
        return 'B'
    elif score >=70:
        return 'C'
    elif score >=60:
        return 'D'
    elif score >=0:
```

```
            return 'E'
        else:
            return 'wrong score.must > 0'
>>> func(120)
'wrong score.must <= 100.'
>>> func(99)
'A'
>>> func(87)
'B'
>>> func(62)
'D'
>>> func(3)
'E'
>>> func(-10)
'wrong score.must > 0'
```

3.2.4 选择结构的嵌套

选择结构可以进行嵌套，语法如下：

```
if 表达式 1:
    语句块 1
    if 表达式 2:
        语句块 2
    else:
        语句块 3
else:
    if 表达式 4:
        语句块 4
```

使用该结构时，一定要严格控制好不同级别代码块的缩进量，因为这决定了不同代码块的从属关系以及业务逻辑是否被正确地实现、是否能够被 Python 正确理解。例如，3.2.3 节中百分制转等级制的示例，作为一种编程技巧，还可以尝试下面的写法：

```
>>> def func(score):
    degree = 'DCBAAE'
    if score > 100 or score < 0:
        return 'wrong score.must between 0 and 100.'
    else:
        index = (score - 60)//10
        if index >= 0:
            return degree[index]
        else:
            return degree[-1]
>>> func(-10)
'wrong score.must between 0 and 100.'
```

```
>>> func(30)
'E'
>>> func(50)
'E'
>>> func(60)
'D'
>>> func(93)
'A'
>>> func(100)
'A'
```

3.2.5 选择结构应用

【例 3-1】 面试资格确认。

```
age = 24
subject = "计算机"
college = "非重点"
if (age > 25 and subject== "电子信息工程") or (college== "重点" and subject== "电子信息工程"
) or (age<= 28 and subject== "计算机"):
    print("恭喜,你已获得我公司的面试机会!")
else:
    print("抱歉,你未达到面试要求")
```

【例 3-2】 用户输入若干个成绩,求所有成绩的总和。每输入一个成绩后询问是否继续输入下一个成绩,回答 yes 就继续输入下一个成绩,回答 no 就停止输入成绩。下面的示例代码使用 Python 2.7.8 编写,读者可以很容易地修改为 Python 3.x 的版本。

```
import types
endFlag = 'yes'
s = 0
while endFlag.lower() == ' yes':
    x = input("请输入一个正整数: ")
    if type(x) == types.IntType and 0<= x<= 100:
        s = s + x
    else:
        print '不是数字或不符合要求'
    endFlag = raw_input('继续输入?(yes or no)')
print '整数之和=', s
```

3.3 循环结构

Python 提供了两种基本的循环结构：while 循环和 for 循环。其中,while 循环一般用于循环次数难以提前确定的情况,当然也可以用于循环次数确定的情况;for 循环一般用于循环次数可以提前确定的情况,尤其适用于枚举或遍历序列或迭代对象中的元素;编程时一

般建议优先考虑使用 for 循环。相同或不同的循环结构之间可以互相嵌套，也可以与选择结构嵌套使用，用来实现更为复杂的逻辑。

while 循环和 for 循环常见的用法为

```
while 表达式：
    循环体
```

和

```
for 变量 in 序列或其他迭代对象：
    循环体
```

另外，while 循环和 for 循环都可以带 else 子句，如果循环因为条件表达式不成立而自然结束（不是因为执行了 break 而结束循环）时则执行 else 结构中的语句，如果循环是因为执行了 break 语句而导致循环提前结束则不执行 else 中的语句。其语法形式为

```
while 表达式：
    循环体
else:
    else 子句
```

和

```
for 取值 in 序列或迭代对象：
    循环体
else:
    else 子句
```

例如，下面的代码演示了带有 else 子句的循环结构，该代码用来计算 1＋2＋3＋…＋99＋100 的结果。

```
>>> s = 0
>>> for i in range(1,101):
    s += i
else:
    print s
5050
```

下面的代码使用 while 循环实现了同样的功能：

```
>>> s = i = 0
>>> while i <= 100:
    s += i
    i += 1
else:
    print s
5050
```

为了优化程序以获得更高的效率和运行速度，在编写循环语句时，应尽量减少循环内部不必要的计算，将与循环变量无关的代码尽可能地提取到循环之外，如果不得不使用多重循

环嵌套时，应尽量减少内层循环中不必要的计算，尽可能地向外提；另外，在循环中应尽量引用局部变量，因为局部变量的查询和访问速度比全局变量略块，在使用模块中的方法时，可以通过将其转换为局部变量来提高运行速度，例如：

```
import time
import math
start = time.time()
for i in xrange(10000000):
    math.sin(i)
print('Time Used:', time.time() - start)
loc_sin = math.sin
start = time.time()
for i in xrange(10000000):
    loc_sin(i)
print('Time Used:', time.time() - start)
```

这段代码演示了模块方法的两种不同调用方式，并比较各自的运行时间，结果为

```
('Time Used:', 4.9059998989105225)
('Time Used:', 4.406000137329102)
```

您应该还记得，在第 1 章还学习过另外一种导入和使用模块成员的方法，把上面的代码修改为

```
import time
from math import sin as sin
start = time.time()
for i in xrange(10000000):
    sin(i)
print('Time Used:', time.time() - start)
loc_sin = sin
start = time.time()
for i in xrange(10000000):
    loc_sin(i)
print('Time Used:', time.time() - start)
```

代码运行结果如下，可以看出，效率也略有提高：

```
('Time Used:', 4.608999967575073)
('Time Used:', 4.4059998989105225)
```

3.4 break 和 continue 语句

break 语句在 while 循环和 for 循环中都可以使用，一般常与选择结构结合使用，以达到在特定条件得到满足时跳出循环的目的。一旦 break 执行语句，将使得整个循环提前结束。continue 语句的作用是终止本次循环，并忽略 continue 之后的所有语句，然后回到循环的顶端，提前进入下一次循环。需要注意的是，过多的 break 和 continue 语句会严重降低程

序的可读性,除非 break 或 continue 语句可以让代码更简单或更清晰,否则不要轻易使用。

下面的代码用来计算小于 100 的最大素数,请注意 break 语句和 else 子句的用法。

```
>>> for n in range(100,1,-1):
    for i in range(2,n):
        if n%i ==0:
            break
    else:
        print n
        break
    97
```

删除上面代码中最后一个 break 语句,则可以用来输出 100 以内的所有素数,例如:

```
>>> for n in range(100,1,-1):
    for i in range(2,n):
        if n%i ==0:
            break
    else:
        print n,
    97 89 83 79 73 71 67 61 59 53 47 43 41 37 31 29 23 19 17 13 11 7 5 3 2
```

在编写循环结构代码时,一定要警惕 continue 语句可能带来的问题,例如下面的代码本意是用来输出 10 以内的奇数:

```
>>> i =1
>>> while i<10:
    if i%2 ==0:
        continue
    print i
    i +=1
```

但是由于代码设计存在问题,从而导致这个循环变成了永不结束的死循环,需要按组合键 Ctrl+C 来强行终止。出现这种情况的原因是,一旦条件表达式 i%2==0 得到满足以后执行 continue 语句,之后的 i+=1 语句将永远不再执行,循环变量永远停留在当前的值,从而使得循环无法结束。上面的代码改成下面这样就不会有问题了:

```
>>> i =1
>>> while i<10:
    if i%2 ==0:
        i +=1
        continue
    print i,
    i +=1
1 3 5 7 9
```

或者修改为下面更为简洁易理解的形式:

```
>>>i =0
>>>while i<10:
    i +=1
    if i%2 ==0:
        continue
    print i,
1 3 5 7 9
```

当然，也可以使用更简洁的 for 循环来实现：

```
>>>for i in range(10):
    if i%2 ==0:
        continue
    print i,
1 3 5 7 9
```

为了充分理解，再修改一下：

```
>>>for i in range(10):
    if i%2 ==0:
        i +=1
        continue
    print i,
1 3 5 7 9
```

在这段代码中，条件语句中 continue 之前的语句 i+=1 并没有起到任何作用。之所以会这样，是因为每次进入循环时的变量 i 已经不再是上一次的变量 i，所以修改其值并不会影响循环的执行。下面的代码很好地描述了这个问题：

```
>>>for i in range(5):
    print id(i),':',i
10416692: 0
10416680: 1
10416668: 2
10416656: 3
10416644: 4
```

3.5 综合运用

本章最后通过几个示例来演示选择结构和循环结构的用法，正如前面所说，这两个结构经常需要互相结合来实现特定的业务逻辑。

【例 3-3】 计算 1+2+3+…+100 的值。

对于这样比较规则的循环，一般优先考虑使用 for 循环，参考 Python 2.7.8，代码如下：

```
s=0
for i in range(1,101):
    s=s +i
```

```
print '1+2+3+…+100 =', s
print '1+2+3+…+100 =', sum(range(1,101))
```

类似的问题也可以使用 while 循环解决，请参考前面 3.4 节的代码。

【例 3-4】 输出序列中的元素。

对于类似元素遍历的问题，一般也优先考虑使用 for 循环，参考代码如下：

```
a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
for i,v in enumerate(a_list):
print '列表的第', i+1, '个元素是：', v
```

对于类似元素遍历的问题，同样也可以使用 while 循环来解决，但是代码要麻烦一些，可读性也较差，例如：

```
>>>a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
>>>i =0
>>>number = len(a_list)
>>>while i <number:
    print '列表的第', i+1, '个元素是：', a_list[i]
    i +=1
列表的第 1个元素是：a
列表的第 2个元素是：b
列表的第 3个元素是：mpilgrim
列表的第 4个元素是：z
列表的第 5个元素是：example
```

【例 3-5】 求 1～100 之间能被 7 整除，但不能同时被 5 整除的所有整数。

该例主要介绍条件表达式的写法，参考代码如下：

```
for i in range(1,101):
    if i %7 ==0 and i %5 !=0:
        print i
```

【例 3-6】 输出“水仙花数”。水仙花数是指一个 3 位的十进制数，其各位数字的立方和等于该数本身。例如，153 是水仙花数，因为 $153=1^3+5^3+3^3$。

```
for i in range(100,1000):
    ge =i %10
    shi =i // 10 %10
    bai =i // 100
    if ge * * 3+shi * * 3+bai * * 3 ==i:
        print i
```

【例 3-7】 求平均分。

```
score = [70, 90, 78, 85, 97, 94, 65, 80]
s =0
for i in score:
    s +=i
```

```
print s*1.0/len(score)
```

如果您对前面学过的序列知识熟悉的话，肯定会想到，其实可以使用下面的内置函数来计算平均分：

```
print sum(score)*1.0/len(score)
```

这里之所以需要将所有元素的和乘以 1.0 转换为浮点数，是因为在 Python 2 中除法运算符/的限制，如果将上面的代码改写为 Python 3 代码，则不再需要这个转换。

【例 3-8】 打印九九乘法表。

该例主要介绍循环结构嵌套用法和循环条件的控制，参考代码如下：

```
for i in range(1,10):
    for j in range(1,i+1):
        print i,'*',j,'=',i*j,'\t',
    print '\n'
```

【例 3-9】 求 200 以内能被 17 整除的最大正整数。

熟练掌握 range()函数的用法，对于很多循环来说可能起到事半功倍的效果，参考代码如下：

```
for i in range(200,0,-1):
    if i%17==0:
        print i
        break
```

【例 3-10】 判断一个数是否为素数。

```
import math
n=input("Input an integer:")
m=int(math.sqrt(n)+1)
for i in range(2,m):
    if n%i==0:
        print 'No'
        break
else:
    print 'Yes'
```

本章知识精要

(1) 几乎所有合法的 Python 表达式都可以作为选择结构和循环结构中的条件表达式。

(2) 选择结构和循环结构往往会互相嵌套使用来实现复杂的业务逻辑。

(3) 应优先考虑使用 for 循环。

(4) 编写循环语句时，应尽量减少内循环中的无关计算。

(5) for 循环和 while 循环都可以带有 else 子句，如果循环因为条件表达式不满足而自然结束时，执行 else 子句中的代码。

（6）除非 break 和 continue 语句可以让代码变得更简单或更清晰，否则请不要轻易使用。

习　　题

1. 分析逻辑运算符 or 的短路求值特性。

2. 编写程序，运行后用户输入 4 位整数作为年份，判断其是否为闰年。如果年份能被 400 整除，则为闰年；如果年份能被 4 整除但不能被 100 整除也为闰年。

3. 编写程序，生成一个包含 50 个随机整数的列表，然后删除其中所有奇数（提示：从后向前删）。

4. 编写程序，生成一个包含 20 个随机整数的列表，然后对其中偶数下标的元素进行降序排列，奇数下标的元素不变（提示：使用切片）。

5. 编写程序，用户从键盘输入小于 1000 的整数，对其进行因式分解。例如，10＝2×5，60＝2×2×3×5。

6. 编写程序，至少使用 2 种不同的方法计算 100 以内所有奇数的和。

7. 编写程序，实现分段函数计算，如表 3-1 所示。

表 3-1　分段函数计算

x	y	x	y
x<0	0	10≤x<20	0.5x−2
0≤x<5	x	20≤x	0
5≤x<10	3x−5		

第 4 章　字符串与正则表达式

最早的字符串编码是美国标准信息交换码 ASCII，仅对 10 个数字、26 个大写英文字母、26 个小写英文字母及一些其他符号进行了编码。ASCII 采用 8 位（即 1 个字节）来对字符进行编码，因此最多只能表示 256 个符号。

随着信息技术的发展和信息交换的需要，各国的文字都需要进行编码，于是分别设计了不同的编码格式，并且编码格式之间有着较大的区别，其中常见的编码有 UTF-8、GB2312、GBK、CP936 等。采用不同的编码格式意味着把同一字符存入文件时，写入的内容可能会不同。其中，UTF-8 编码是国际通用的编码，以 1 个字节表示英语字符（兼容 ASCII），以 3 个字节表示中文及其他语言，UTF-8 对全世界所有国家需要用到的字符进行了编码。

GB2312 是中国制定的中文编码，使用 1 个字节表示英语，2 个字节表示中文；GBK 是 GB2312 的扩充，而 CP936 是微软公司在 GBK 基础上完成的编码。GB2312、GBK 和 CP936 都是使用 2 个字节表示中文，UTF-8 使用 3 个字节表示中文。在众多编码方案中，Unicode 是不同编码格式之间进行互相转换的基础。

在 Windows 平台上，input()函数从键盘输入的字符串默认为 GBK 编码，而 Python 程序中的字符串编码则使用＃coding 显式地指定，常用的方式如下：

```
#coding=utf-8
#coding:GBK
#-*-coding:utf-8-*-
```

Python 2 对中文支持不够，因此常常需要在不同的编码之间互相转换，例如，下面代码是在 Python 2.7.8 环境下执行的结果：

```
>>> s1='中国'
>>> s1
'\xd6\xd0\xb9\xfa'
>>> len(s1)
4
>>> s2=s1.decode('GBK')
>>> s2
u'\u4e2d\u56fd'
>>> len(s2)
2
>>> s3=s2.encode('UTF-8')
>>> s3
'\xe4\xb8\xad\xe5\x9b\xbd'
>>> len(s3)
6
>>>print s1,s2,s3
```

```
中国 中国 中国
```

Python 3 中则完全支持中文，无论是一个数字、英文字母，还是一个汉字，都按一个符号对待和处理。例如，在 Python 3.4.2 环境中执行下面的代码：

```
>>> s = '中国山东烟台'
>>> len(s)
6
>>> s = 'SDIBT'
>>> len(s)
5
>>> s = '中国山东烟台 SDIBT'
>>> len(s)
11
```

4.1 字 符 串

在 Python 中，字符串属于不可变序列类型，使用单引号、双引号、三单引号或三双引号作为界定符，并且不同的界定符之间可以互相嵌套。除了支持序列通用方法(包括比较、计算长度、元素访问和分片等操作)以外，字符串类型还支持一些特有的操作方法，例如格式化操作、字符串查找、字符串替换等。但由于字符串属于不可变序列，不能对字符串对象进行元素增加、修改与删除等操作。字符串对象提供的 replace()和 translate()方法并不是对原字符串直接进行修改替换，而是返回一个修改替换后的结果字符串，并不对原字符串做任何改动。

Python 支持字符串驻留机制，即：对于短字符串，将其赋值给多个不同的对象时，内存中只有一个副本，多个对象共享该副本。这一点不适用于长字符串，即长字符串不遵守驻留机制，下面的代码演示了短字符串和长字符串在这方面的区别：

```
>>> a = '1234'
>>> b = '1234'
>>> id(a) == id(b)
True
>>> a = '1234' * 50
>>> b = '1234' * 50
>>> id(a) == id(b)
False
```

如果需要判断一个变量 s 是否为字符串，应使用 isinstance(s, basestring)。在 Python 3 之前，字符串有 str 和 Unicode 两种，其基类都是 basestring，在 Python 3 之后合二为一了。在 Python 3 中，程序源文件默认为 UTF-8 编码，全面支持中文，字符串对象不再有 encode 和 decode 方法。甚至在 Python 3 中可以使用中文作为变量名。下面的代码演示了 Python 2.7.8 中的字符串类型：

```
>>> import types
```

```
>>> types.StringType
<type 'str'>
>>> basestring
<type 'basestring'>
>>> s = 'hello world'
>>> isinstance(s,basestring)
True
>>> type(s)
<type 'str'>
>>> type(s) == types.StringType
True
>>> ss = u'hello world'
>>> type(ss)
<type 'unicode'>
>>> isinstance(ss,basestring)
True
>>> type(ss) == types.UnicodeType
True
>>> type(ss) == types.StringType
False
```

4.1.1 字符串格式化

如果需要将其他类型数据转换为字符串或另一种数字格式,或者嵌入其他字符串或模板中再进行输出,就需要用到字符串格式化。Python 中字符串格式化的格式如图 4-1 所示,%符号之前的部分为格式字符串,%之后的部分为需要进行格式化的内容。

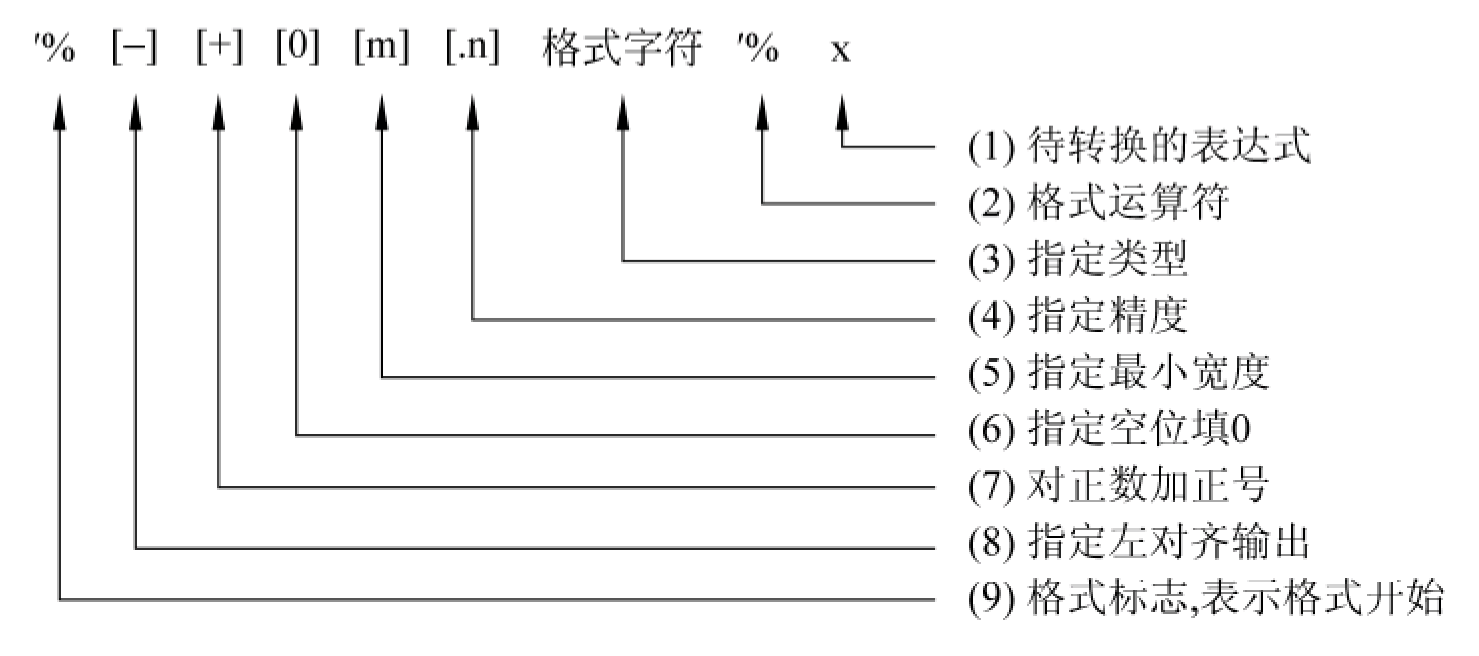


图 4-1 字符串格式化

与其他语言一样,Python 支持大量的格式字符,常见的格式字符如表 4-1 所示。

表 4-1 常见的格式字符

格式字符	说　明	格式字符	说　明
%s	字符串(采用 str()的显示)	%c	单个字符
%r	字符串(采用 repr()的显示)	%b	二进制整数

续表

格式字符	说　　明	格式字符	说　　明
%d	十进制整数	%E	指数(基底写为 E)
%i	十进制整数	%f、%F	浮点数
%o	八进制整数	%g	指数(e)或浮点数(根据显示长度)
%x	十六进制整数	%G	指数(E)或浮点数(根据显示长度)
%e	指数(基底写为 e)	%%	字符%

下面的代码简单演示了字符串格式化的用法：

```
>>>x =1235
>>>so ="%o" %x
>>>so
"2323"
>>>sh ="%x" %x
>>>sh
"4d3"
>>>se ="%e" %x
>>>se
"1.235000e+03"
>>>chr(ord("3")+1)
"4"
>>>"%s"%65              #类似于 str()
"65"
>>>"%s"%65333
"65333"
>>>'%d,%c'%(65,65)      #使用元组对字符串进行格式化,按位置进行对应
'65,A'
>>>"%d"%"555"           #试图将字符串转换为整数进行输出,抛出异常
Traceback (most recent call last):
    File "<pyshell#19>", line 1, in <module>
        "%d"%"555"
TypeError: %d format: a number is required, not str
>>>int('555')           #可以使用 int()函数将合法的数字字符串转换为整数
555
>>>'%s'%[1,2,3]
'[1, 2, 3]'
>>>str((1,2,3))         #可以使用 str()函数将任意类型数据转换为字符串
'(1, 2, 3)'
>>>str([1,2,3])
'[1, 2, 3]'
```

除了上面介绍的字符串格式化方法,目前 Python 社区更推荐使用 format()方法进行

格式化,该方法更加灵活,不仅可以使用位置进行格式化,还支持使用与位置无关的名字来进行格式化,并且支持序列解包格式化字符串,为程序员提供了方便。例如:

```
>>>print "The number {0:,} in hex is: {0:#x}, the number {1} in oct is {1:#o}".format(5555,55)
The number 5,555 in hex is: 0x15b3, the number 55 in oct is 0o67
>>>print "The number {1:,} in hex is: {1:#x}, the number {0} in oct is {0:#o}".format(5555,55)
The number 55 in hex is: 0x37, the number 5555 in oct is 0o12663
>>>print "my name is {name}, my age is {age}, and my QQ is {qq}".format(name =
"Dong Fuguo", age = 37, qq = "306467355")
my name is Dong Fuguo, my age is 37, and my QQ is 306467355
>>>position = (5,8,13)
>>>print "X:{0[0]};Y:{0[1]};Z:{0[2]}".format(position)
X:5;Y:8;Z:13
>>>weather = [("Monday","rain"), ("Tuesday","sunny"), ("Wednesday", "sunny"),
("Thursday","rain"), ("Friday","Cloudy")]
>>>formatter = "Weather of '{0[0]}' is '{0[1]}'".format
>>>for item in map(formatter,weather):
    print item
Weather of 'Monday' is 'rain'
Weather of 'Tuesday' is 'sunny'
Weather of 'Wednesday' is 'sunny'
Weather of 'Thursday' is 'rain'
Weather of 'Friday' is 'Cloudy'
```

4.1.2 字符串常用方法

字符串是非常重要的数据类型,Python 提供了大量的函数支持字符串操作,本节通过几个示例来演示部分函数的用法,可以使用 dir("")查看所有字符串操作函数列表,并使用内置函数 help()查看每个函数的帮助。除了本节介绍的字符串处理函数,部分 Python 内置函数也支持对字符串的操作,例如用来计算序列长度的 len()方法,用来比较序列大小的 cmp()方法等,因为字符串也是 Python 序列的一种。

1. find()

该函数用来查找一个字符串在另一个字符串指定范围(默认是整个字符串)中首次出现的位置,如果不存在则返回−1。

```
>>>s= "apple,peach,banana,peach,pear"
>>>s.find("peach")
6
>>>s.find("peach",7)
19
>>>s.find("peach",7,20)
-1
```

2. split()

该函数用来以指定字符为分隔符,将字符串分割成多个字符串,并返回包含分割结果的

列表。

```
>>> s="apple,peach,banana,pear"
>>> li=s.split(",")
>>> li
["apple", "peach", "banana", "pear"]
>>> s ="2014-10-31"
>>> t=s.split("-")
>>> print t
['2014', '10', '31']
>>> print map(int, t)
[2014, 10, 31]
```

如果不指定分隔符，则字符串中的任何空白符号(包括空格、换行符和制表符等)都将被认为是分隔符，返回包含最终分割结果的列表。

```
>>> s = 'hello world \n\n My name is Dong '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Dong']
>>> s = '\n\nhello world \n\n\n My name is Dong '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Dong']
>>> s = '\n\nhello\t\t world \n\n\n My name\t is Dong '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Dong']
```

该方法还允许指定最大分割次数，例如：

```
>>> s = '\n\nhello\t\t world \n\n\n My name is Dong '
>>> s.split(None,1)
['hello', 'world \n\n\n My name is Dong ']
>>> s.split(None,2)
['hello', 'world', 'My name is Dong ']
>>> s.split(None,5)
['hello', 'world', 'My', 'name', 'is', 'Dong ']
>>> s.split(None,6)
['hello', 'world', 'My', 'name', 'is', 'Dong']
```

3. join()

与 split()方法相反，该函数用来将列表中多个字符串进行连接，并在相邻两个字符串之间插入指定字符。

```
>>> li=["apple", "peach", "banana", "pear"]
>>> sep=","
>>> s=sep.join(li)
>>> s
"apple,peach,banana,pear"
```

使用运算符+也可以连接字符串，但效率较低，应优先使用 join()方法。下面的代码演示了两者之间速度的差异：

```
import timeit
strlist=['This is a long string that will not keep in memory.' for n in xrange(100)]
def use_join():
    return ''.join(strlist)
def use_plus():
    result=''
    for strtemp in strlist:
        result=result+strtemp
    return result
if __name__=='__main__':
    times=1000
    jointimer=timeit.Timer('use_join()','from __main__ import use_join')
    print 'time for join:',jointimer.timeit(number=times)
    plustimer=timeit.Timer('use_plus()','from __main__ import use_plus')
    print 'time for plus:',plustimer.timeit(number=times)
```

该代码分别使用 join()函数和+对 100 个字符串进行连接，并重复运行 1000 次，然后输出每种方法所使用的时间，运行结果为

```
time for join: 0.00395874865103
time for plus: 0.0260573301694
```

4. lower()、upper()、capitalize()、title()和 swapcase()

这几个函数分别用来将字符串转换为小写字符串、将字符串转换为大写字符串、将字符串首字母变为大写、将每个单词的首字母变为大写以及大小写互换。

```
>>> s="What is Your Name?"
>>> s2=s.lower()
>>> s2
"what is your name?"
>>> s.upper()
"WHAT IS YOUR NAME?"
>>> s2.capitalize()
"What is your name?"
>>> s.title()
'What Is Your Name?'
>>> s.swapcase()
'wHAT IS yOUR nAME?'
```

5. replace()

该函数用来替换字符串中指定字符或子字符串的所有重复出现，每次只能替换一个字符或一个子字符串的重复出现。

```
>>> s="中国,中国"
>>> print s
中国,中国
>>> s2=s.replace("中国", "中华人民共和国")
>>> print s2
中华人民共和国,中华人民共和国
```

6. maketrans()和 translate()

maketrans()函数用来生成字符映射表,而 translate()则按映射表关系转换字符串并替换其中的字符,使用这两个函数的组合可以同时处理多个不同的字符,replace()则无法满足这一要求。下面的代码演示了这两个函数的用法,当然您还可以定义自己的字符映射表,然后用来对字符串进行加密。

```
>>> import string
#将字符"abcdef123"一一对应地转换为"uvwxyz@#$"
>>> table=string.maketrans("abcdef123","uvwxyz@#$")
>>> s="Python is a greate programming language. I like it!"
>>> s.translate(table)
"Python is u gryuty progrumming lunguugy. I liky it!"
>>> s.translate(table,"gtm")      #第二个参数表示要删除的字符
"Pyhon is u ryuy proruin lunuuy. I liky i!"
```

7. strip()和 rstrip()

该函数用来删除两端(或右端)的空白字符或连续的指定字符。

```
>>> s=" abc "
>>> s2=s.strip()
>>> s2
"abc"
>>> "aaaassddf".strip("a")
"ssddf"
>>> "aaaassddf".strip("af")
"ssdd"
>>> "aaaassddfaaa".rstrip("a")
'aaaassddf'
```

8. eval()

内置函数 eval()尝试把任意字符串转化为 Python 表达式并进行求值。

```
>>> eval("3+4")
7
>>> a =3
>>> b =5
>>> eval('a+b')
8
>>> import math
```

```
>>> eval('help(math.sqrt)')
Help on built-in function sqrt in module math:
sqrt(…)
    sqrt(x)
    Return the square root of x.
>>> eval('math.sqrt(3)')
1.7320508075688772
>>> eval('aa')
Traceback (most recent call last):
    File "<pyshell#3>", line 1, in <module>
        eval('aa')
    File "<string>", line 1, in <module>
NameError: name 'aa' is not defined
```

使用 eval()时要注意的一个问题是，它可以计算任意合法表达式的值，如果用户巧妙地构造输入，可以执行任意外部程序，例如下面的代码运行后可以启动记事本程序：

```
>>> a = input("Please input a value:")
Please input a value:"__import__('os').startfile(r'C:\windows\\notepad.exe')"
>>> eval(a)
```

是不是非常危险啊？如果您觉得这没什么的话，再执行下面的代码试试，然后看看当前工作目录中多了什么，当然您可以调用命令来删除这个文件夹或其他文件。

```
>>> eval("__import__('os').system('md testtest')")
```

9. 关键字 in

与列表、元组、字典、集合一样，也可以使用关键字 in 和 not in 来判断一个字符串是否出现在另一个字符串中。

```
>>> "a" in "abcde"
True
>>> 'ab' in 'abcde'
True
>>> "j" in "abcde"
False
```

10. startswith()和 endswith()

这两个函数用来判断字符串是否以指定字符串开始或结束。在 2.1.8 节中介绍列表推导式时用到过，请自行翻阅。

11. isalnum()、isalpha()和 isdigit()

用来测试字符串是否为数字或字母、是否为字母、是否为数字字符。

```
>>> '1234abcd'.isalnum()
True
>>> '1234abcd'.isalpha()
False
```

```
>>> '1234abcd'.isdigit()
False
>>> 'abcd'.isalpha()
True
>>> '1234.0'.isdigit()
False
>>> '1234'.isdigit()
True
```

4.1.3 字符串常量

在 string 模块中定义了多个字符串常量，包括数字字符、标点符号、英文字母、大写字母和小写字符等，用户可以直接使用这些常量。

```
>>> import string          # Python 2.7.8
>>> string.digits
'0123456789'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_'{|}~ '
>>> string.letters
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?
@[\\]^_'{|}~ \t\n\r\x0b\x0c'
>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

4.2 正则表达式

正则表达式是字符串处理的有力工具和技术，正则表达式使用预定义的特定模式去匹配一类具有共同特征的字符串，主要用于处理字符串，可以快速、准确地完成复杂的查找、替换等处理要求。

Python 中，re 模块提供了正则表达式操作所需要的功能。本节中首先介绍正则表达式的基础知识，然后介绍 re 模块提供的正则表达式函数与对象的用法。

4.2.1 正则表达式元字符

正则表达式由元字符及其不同组合来构成，通过巧妙地构造正则表达式可以匹配任意类型字符串。常用的正则表达式元字符如表 4-2 所示。

表 4-2 正则表达式常用元字符

元字符	功 能 说 明

<table>
<tr><th>元字符</th><th>功 能 说 明</th></tr>
<tr><td>.</td><td>匹配除换行符以外的任意单个字符</td></tr>
<tr><td>*</td><td>匹配位于 * 之前的 0 个或多个字符</td></tr>
<tr><td>+</td><td>匹配位于+之前的一个或多个字符</td></tr>
<tr><td>|</td><td>匹配位于|之前或之后的字符</td></tr>
<tr><td>^</td><td>匹配行首，匹配以^后面的字符开头的字符串</td></tr>
<tr><td>$</td><td>匹配行尾，匹配以 $ 之前的字符结束的字符串</td></tr>
<tr><td>?</td><td>匹配位于“?”之前的 0 个或 1 个字符</td></tr>
<tr><td>\</td><td>表示位于\之后的为转义字符</td></tr>
<tr><td>[]</td><td>匹配位于[]中的任意一个字符</td></tr>
<tr><td>-</td><td>用在[]之内用来表示范围</td></tr>
<tr><td>()</td><td>将位于()内的内容作为一个整体来对待</td></tr>
<tr><td>{}</td><td>按{}中的次数进行匹配</td></tr>
<tr><td>\b</td><td>匹配单词头或单词尾</td></tr>
<tr><td>\B</td><td>与\b 含义相反</td></tr>
<tr><td>\d</td><td>匹配任何数字，相当于[0-9]</td></tr>
<tr><td>\D</td><td>与\d 含义相反</td></tr>
<tr><td>\s</td><td>匹配任何空白字符</td></tr>
<tr><td>\S</td><td>与\s 含义相反</td></tr>
<tr><td>\w</td><td>匹配任何字母、数字以及下划线，相当于[a-z A-Z 0-9_]</td></tr>
<tr><td>\W</td><td>与\w 含义相反</td></tr>
</table>

如果以\开头的元字符与转义字符相同，则需要使用\\，或者使用原始字符串，即在字符串前加上字符 r。

具体应用时，可以单独使用某种类型的元字符，但处理复杂字符串时，经常需要将多个正则表达式元字符进行组合，下面给出了几个简单的示例。

(1) 最简单的正则表达式是普通字符串，可以匹配自身。

(2) '[pjc]ython'可以匹配'python'、'jython'、'cython'。

(3) '[a-zA-Z0-9]'可以匹配一个任意大小写字母或数字。

(4) '[^abc]'可以匹配任意一个除'a'、'b'、'c'之外的字符。

(5) 'python|perl'或'p(ython|erl)'都可以匹配'python'或'perl'。

(6) 子模式后面加上问号表示可选。r'(http://)? (www\.)? python\.org'只能匹配'http://www.python.org'、'http://python.org'、'www.python.org'和'python.org'。

(7) '^http'只能匹配所有以'http'开头的字符串。

(8) (pattern)*：允许模式重复 0 次或多次。

(9) (pattern)+：允许模式重复 1 次或多次。

(10) (pattern){m,n}：允许模式重复 m～n 次。

在具体构造正则表达式时，要注意可能会发生的错误，尤其是涉及特殊字符时。例如，下面这段代码在第 1 章中已经出现过，作用是用来匹配 Python 程序中的运算符，但是有些运算符与正则表达式的元字符相同，如果处理不当则会造成错误：

```
>>> import re
>>> symbols = [',', '+', '-', '*', '/', '//', '**', '>>', '<<', '+=', '-=', '*=', '/=']
>>> for i in symbols:
    patter = re.compile(r'\s*'+i+r'\s*')
Traceback (most recent call last):
    File "<pyshell#11>", line 2, in <module>
        patter = re.compile(r'\s*'+i+r'\s*')
    File "C:\python27\lib\re.py", line 190, in compile
        return _compile(pattern, flags)
    File "C:\python27\lib\re.py", line 244, in _compile
        raise error, v # invalid expression
error: multiple repeat
>>> for i in symbols:
    patter = re.compile(r'\s*'+re.escape(i)+r'\s*')
正常执行
```

4.2.2 re 模块主要方法

在 Python 中，主要使用 re 模块来实现正则表达式的操作。该模块的常用方法如表 4-3 所示，具体使用时，既可以直接使用 re 模块的方法进行字符串处理，也可以将模式编译为正则表达式对象，然后使用正则表达式对象的方法来操作字符串。

表 4-3 re 模块常用方法

方　法	功 能 说 明
compile(pattern[,flags])	创建模式对象
search(pattern,string[,flags])	在整个字符串中寻找模式，返回 match 对象或 None
match(pattern,string[,flags])	从字符串的开始处匹配模式，返回 match 对象或 None
findall(pattern,string[,flags])	列出字符串中模式的所有匹配项
split(pattern,string[,maxsplit=0])	根据模式匹配项分割字符串
sub(pat,repl,string[,count=0])	将字符串中所有 pat 的匹配项用 repl 替换
escape(string)	将字符串中所有特殊正则表达式字符转义

其中函数参数 flags 的值可以是 re. I(忽略大小写)、re. L、re. M(多行匹配模式)、re. S(使元字符“.”匹配任意字符，包括换行符)、re. U(匹配 Unicode 字符)、re. X(忽略模式中的空格，并可以使用#注释)的不同组合(使用|进行组合)。

4.2.3 直接使用 re 模块的方法

可以直接使用 re 模块的方法来实现正则表达式操作，本节通过几个具体的示例来简单

演示其用法。

```
>>> import re
>>> text = 'alpha. beta…gamma delta'
>>> re.split('[\. ]+',text)
['alpha', 'beta', 'gamma', 'delta']
>>> re.split('[\. ]+',text,maxsplit=2)        #分割 2 次
['alpha', 'beta', 'gamma delta']
>>> re.split('[\. ]+',text,maxsplit=1)        #分割 1 次
['alpha', 'beta…gamma delta']
>>> pat = '[a-zA-Z]+'
>>> re.findall(pat,text)                      #查找所有单词
['alpha', 'beta', 'gamma', 'delta']
>>> pat = '{name}'
>>> text = 'Dear {name}…'
>>> re.sub(pat,'Mr.Dong',text)                #字符串替换
'Dear Mr.Dong…'
>>> s = 'a s d'
>>> re.sub('a|s|d','good',s)                  #字符串替换
'good good good'
>>> re.escape('http://www.python.org')        #字符串转义
'http\\:\\/\\/www\\.python\\.org'
>>> print re.match('done|quit','done')        #匹配成功
<_sre.SRE_Match object at 0x00B121A8>
>>> print re.match('done|quit','done!')       #匹配成功
<_sre.SRE_Match object at 0x00B121A8>
>>> print re.match('done|quit','doe!')        #匹配不成功
None
>>> print re.match('done|quit','d!one!')      #匹配不成功
None
```

下面的代码使用不同的方法删除字符串中多余的空格，如果遇到连续多个空格则只保留一个。

```
>>> import re
>>> s = 'aaa    bb    c d e    fff'
>>> re.sub('\s+', ' ',s)                      #直接使用 re 模块的字符串替换方法
'aaa bb c d e fff '
>>> re.split('[\s]+', s)
['aaa', 'bb', 'c', 'd', 'e', 'fff', '']
>>> re.split('[\s]+',s.strip())
['aaa', 'bb', 'c', 'd', 'e', 'fff']
>>> ' '.join(re.split('[\s]+',s.strip()))
'aaa bb c d e fff'
>>> ' '.join(re.split('\s+',s.strip()))
'aaa bb c d e fff'
```

```
>>> re.sub('\s+',' ',s.strip())
'aaa bb c d e fff'
>>> s.split()                          #也可以不使用正则表达式
['aaa', 'bb', 'c', 'd', 'e', 'fff']
>>> ' '.join(s.split())
'aaa bb c d e fff'
```

下面的代码使用以\开头的元字符来实现字符串的特定搜索。

```
>>> import re
>>> example = 'ShanDong Institute of Business and Technology'
>>> re.findall('\\ba.+?\\b',example)       #以 a 开头的完整单词
['and']
>>> re.findall('\\Bo.+?\\b',example)       #不以 o 开头且含有 o 字母的单词剩余部分
['ong', 'ology']
>>> re.findall('\\b\w.+?\\b',example)      #所有单词
['ShanDong', 'Institute', 'of', 'Business', 'and', 'Technology']
>>> re.findall(r'\b\w.+?\b',example)       #使用原始字符串,减少需要输入的符号数量
['ShanDong', 'Institute', 'of', 'Business', 'and', 'Technology']
>>> re.findall('\d\.\d\.\d','Python 2.7.8')  #查找并返回 x.x.x 形式的数字
['2.7.8']
>>> re.split('\s',example)                 #使用任何空白字符分割字符串
['ShanDong', 'Institute', 'of', 'Business', 'and', 'Technology']
```

4.2.4 使用正则表达式对象

首先使用 re 模块的 compile()方法将正则表达式编译生成正则表达式对象，然后再使用正则表达式对象提供的方法进行字符串处理，使用编译后的正则表达式对象可以提高字符串处理速度。

正则表达式对象的 match(string[,pos[,endpos]])方法用于在字符串开头或指定位置进行搜索，模式必须出现在字符串开头或指定位置；search(string[,pos[,endpos]])方法用于在字符串整个指定范围中进行搜索；findall(string[,pos[,endpos]])方法用于在字符串中查找所有符合正则表达式的字符串并以列表形式返回。

```
import re
>>> example = 'ShanDong Institute of Business and Technology'
>>> pattern = re.compile(r'\bB\w+\b')        #以 B 开头的单词
>>> pattern.findall(example)
['Business']
>>> pattern = re.compile(r'\w+g\b')          #以 g 结尾的单词
>>> pattern.findall(example)
['ShanDong']
>>> pattern = re.compile(r'\b[a-zA-Z]{3}\b')  #查找 3 个字母长的单词
>>> pattern.findall(example)
['and']
>>> pattern.match(example)                   #从字符串开头开始匹配,不成功,没有返回值
```

```
>>>pattern.search(example)                #在整个字符串中搜索,成功
<_sre.SRE_Match object at 0x01228EC8>
>>>pattern=re.compile(r'\b\w* a\w* \b')   #查找所有含有字母 a 的单词
>>>pattern.findall(example)
['ShanDong', 'and']
```

正则表达式对象的 sub(repl,string[,count=0])和 subn(repl,string[,count=0])方法用来实现字符串替换功能。

```
>>>example='''Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.'''
pattern=re.compile(r'\bb\w* \b',re.I)
>>>print pattern.sub('* ',example)        #将以字母 b 和 B 开头的单词替换为 *
 *  is *  than ugly.
Explicit is *  than implicit.
Simple is *  than complex.
Complex is *  than complicated.
Flat is *  than nested.
Sparse is *  than dense.
Readability counts.
>>>print pattern.sub('* ',example,1)      #只替换一次
 *  is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
>>>pattern=re.compile(r'\bb\w* \b')
>>>print pattern.sub('* ',example,1)      #将第一个以字母 b 开头的单词替换为 *
Beautiful is *  than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
```

正则表达式对象的 split(string[,maxsplit =0])方法用来实现字符串分割。

```
>>>example=r'one,two,three.four/file\six?seven[eight]nine|ten'
>>>pattern=re.compile(r'[,./\\?[\]\|]')
```

```
>>>pattern.split(example)
['one', 'two', 'three', 'four', 'file', 'six', 'seven', 'eight', 'nine', 'ten']
>>>example =r'one1two2three3four4file5six6seven7eight8nine9ten'
>>>pattern =re.compile(r'\d+ ')
>>>pattern.split(example)
['one', 'two', 'three', 'four', 'file', 'six', 'seven', 'eight', 'nine', 'ten']
>>>example =r'one two three four,file.six.seven,eight,nine9ten'
>>>pattern =re.compile(r'[\s,.\d]+ ')
>>>pattern.split(example)
['one', 'two', 'three', 'four', 'file', 'six', 'seven', 'eight', 'nine', 'ten']
```

4.2.5 子模式与 match 对象

使用圆括号"()"表示一个子模式，圆括号内的内容作为一个整体出现，例如(red)+可以匹配 redred、redredred 等多个重复 red 的情况。

```
>>>telNumber ='''Suppose my Phone No. is 0535-1234567,
yours is 010-12345678, his is 025-87654321.'''
>>>pattern =re.compile(r'(\d{3,4})-(\d{7,8})')
>>>pattern.findall(telNumber)
[('0535', '1234567'), ('010', '12345678'), ('025', '87654321')]
```

正则表达式对象的 match()方法和 search()方法匹配成功后返回 match 对象。match 对象的主要方法有 group()、groups()、groupdict()、start()、end()和 span()等。

```
import re
telNumber ='''Suppose my Phone No. is 0535-1234567, yours is 010-12345678, his is 025-
87654321.'''
pattern =re.compile(r'(\d{3,4})-(\d{7,8})')
index =0
while True:
    matchResult =pattern.search(telNumber, index)
    if not matchResult:
        break
    print '-' * 30
    print 'Success:'
    for i in range(3):
        print 'Searched content:', matchResult.group(i),\
        ' Start from:',matchResult.start(i),'End at:',matchResult.end(i),\
            ' Its span is:', matchResult.span(i)
index =matchResult.end(2)
```

上面程序的运行结果如下：

```
------------------------------
Success:
Searched content: 0535-1234567 Start from: 24 End at: 36 Its span is: (24, 36)
```

```
Searched content: 0535 Start from: 24 End at: 28 Its span is: (24, 28)
Searched content: 1234567 Start from: 29 End at: 36 Its span is: (29, 36)
------------------------------
Success:
Searched content: 010-12345678 Start from: 47 End at: 59 Its span is: (47, 59)
Searched content: 010 Start from: 47 End at: 50 Its span is: (47, 50)
Searched content: 12345678 Start from: 51 End at: 59 Its span is: (51, 59)
------------------------------
Success:
Searched content: 025-87654321 Start from: 68 End at: 80 Its span is: (68, 80)
Searched content: 025 Start from: 68 End at: 71 Its span is: (68, 71)
Searched content: 87654321 Start from: 72 End at: 80 Its span is: (72, 80)
```

使用子模式扩展语法可以实现更加复杂的字符串处理,常用的子模式扩展语法如表 4-4 所示。

表 4-4 子模式扩展语法

语 法	功 能 说 明
(?P<groupname>)	为子模式命名
(?iLmsux)	设置匹配标志,可以是几个字母的组合,每个字母含义与编译标志相同
(?:…)	匹配但不捕获该匹配的子表达式
(?P=groupname)	表示在此之前的命名为 groupname 的子模式
(?#…)	表示注释
(?=…)	用于正则表达式之后,表示如果=后的内容在字符串中出现则匹配,但不返回=之后的内容
(?!…)	用于正则表达式之后,表示如果!后的内容在字符串中不出现则匹配,但不返回!之后的内容
(?<=…)	用于正则表达式之前,与(?=…)含义相同
(?<!…)	用于正则表达式之前,与(?!…)含义相同

下面通过几个示例来演示子模式扩展语法的应用。

```
>>> import re
>>> exampleString = '''There should be one—and preferably only one—obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than right now.'''
>>> pattern = re.compile(r'(?<=\w\s)never(?=\s\w)')   #查找不在句子开头和结尾的单词
>>> matchResult = pattern.search(exampleString)
>>> matchResult.span()
(172, 177)
>>> pattern = re.compile(r'(?<=\w\s)never')       #查找位于句子末尾的单词
>>> matchResult = pattern.search(exampleString)
>>> matchResult.span()
```

```
(156, 161)
>>>pattern = re.compile(r'(?:is\s)better(\sthan)')      #查找前面是 is 的 better than 组合
>>>matchResult = pattern.search(exampleString)
>>>matchResult.span()
(141, 155)
>>>matchResult.group(0)                                  #0 表示整个模式
'is better than'
>>>matchResult.group(1)
' than'
>>>pattern = re.compile(r'\b(?i)n\w+\b')                 #查找以 n 或 N 字母开头的所有单词
>>>index = 0
>>>while True:
    matchResult = pattern.search(exampleString,index)
    if not matchResult:
        break
    print matchResult.group(0), ':', matchResult.span(0)
    index = matchResult.end(0)
not : (92, 95)
Now : (137, 140)
never : (156, 161)
never : (172, 177)
now : (205, 208)
>>>pattern = re.compile(r'(?<!not\s)be\b')               #查找前面没有单词 not 的单词 be
>>>index = 0
>>>while True:
    matchResult = pattern.search(exampleString,index)
    if not matchResult:
        break
    print matchResult.group(0), ':', matchResult.span(0)
    index = matchResult.end(0)
be : (13, 15)
>>>exampleString[13:20]                                  #验证一下结果是否正确
'be one- '
>>>pattern = re.compile(r'(\b\w* (?P<f>\w+) (?P=f)\w* \b)')       #查找具有连续相同字母的单词
>>>index = 0
>>>while True:
    matchResult = pattern.search(exampleString, index)
    if not matchResult:
        break
    print matchResult.group(0),':',matchResult.group(2)
    index = matchResult.end(0) + 1
unless: s
better: t
better: t
>>>s
```

```
'aabc abcd abbcd abccd abcdd'
>>>p = re.compile(r'(\b\w* (?P<f>\w+)(?P=f)\w* \b)')
>>>p.findall(s)
[('aabc', 'a'), ('abbcd', 'b'), ('abccd', 'c'), ('abcdd', 'd')]
```

4.2.6 正则表达式综合运用

在本章的最后，我们通过一个在实际开发中非常有用的案例来演示字符串处理以及正则表达式的使用。在本例中，除了前面章节中介绍的列表、字典等知识，还用到了后面的知识，例如文件操作，请自行查阅。将本案例代码存为文件 FindIdentifiersFromPyFile.py，运行时按照提示输入要检测的 Python 源程序文件，该程序会自动提取出 Python 源文件中所有的类名、函数名以及各种变量名。当然，您可以很容易地修改本程序以实现您需要的更为复杂的业务逻辑。

```
import re
import os

classes = {}
functions = []
variables = {'normal':{}, 'parameter':{}, 'infor':{}}

def _identifyClassNames(index, line):
    '''parameter index is the line number of line,
        parameter line is a line of code of the file to check'''
    pattern = re.compile(r'(?<=class\s)\w+(?=.*?:)')
    matchResult = pattern.search(line)
    if not matchResult:
        return
    className = matchResult.group(0)
    classes[className] = classes.get(className, [])
    classes[className].append(index)

def _identifyFunctionNames(index, line):
    pattern = re.compile(r'(?<=def\s)(\w+)\((.*?)\)(?=:)')
    matchResult = pattern.search(line)
    if not matchResult:
        return
    functionName = matchResult.group(1)
    functions.append((functionName, index))
    parameters = matchResult.group(2).split(r', ')
    if parameters[0] == '':
        return
    for v in parameters:
        variables['parameter'][v] = variables['parameter'].get(v, [])
        variables['parameter'][v].append(index)
```

```
def _identifyVariableNames(index, line):
    # find normal variables, including the case: a, b = 3, 5
    pattern = re.compile(r'\b(.*?)(?=\s=)')
    matchResult = pattern.search(line)
    if matchResult:
        vs = matchResult.group(1).split(r', ')
        for v in vs:
            # consider the case 'if variable == value'
            if 'if ' in v:
                v = v.split()[1]
            # consider the case: 'a[3] = 3'
            if '[' in v:
                v = v[0:v.index('[')]
            variables['normal'][v] = variables['normal'].get(v, [])
            variables['normal'][v].append(index)

    # find the variables in for statements
    pattern = re.compile(r'(?<=for\s)(.*?)(?=\sin)')
    matchResult = pattern.search(line)
    if matchResult:
        vs = matchResult.group(1).split(r', ')
        for v in vs:
            variables['infor'][v] = variables['infor'].get(v, [])
            variables['infor'][v].append(index)

def output():
    print '=' * 30
    print 'The classes' names and their line numbers are:'
    for key, value in classes.items():
        print key, ':', value

    print '=' * 30
    print 'The functions' names and their line numbers are:'
    for i in functions:
        print i[0], ':', i[1]

    print '=' * 30
    print 'The normal variable names and their line numbers are:'
    for key, value in variables['normal'].items():
        print key, ':', value
    print '-' * 20
    print 'The parameter names and their line numbers in functions are:'
    for key, value in variables['parameter'].items():
        print key, ':', value
```

```
    print '-' * 20
    print 'The variable names and their line numbers in for statements are:'
    for key, value in variables['infor'].items():
        print key, ':', value

#suppose the lines of comments less than 50
def comments(index):
    for i in range(50):
        line = allLines[index + i].strip()
        if line.endswith('"""') or line.endswith("'''"):
            return i+1

if __name__ == '__main__':
    fileName = input('Please input the file name (including the full path if neccesary:')
    if not os.path.isfile(fileName):
        print 'Your input is not a file.'
    if not fileName.endswith('.py'):
        print 'Sorry. I can only check Python source file.'
    allLines = []
    with open(fileName, 'r') as fp:
        allLines = fp.readlines()

    index = 0
    totalLen = len(allLines)
    while index < totalLen:
        line = allLines[index]
        #strip the blank characters at both end of line
        line = line.strip()
        #ignore the comments starting with '#'
        if line.startswith('#'):
            index += 1
            continue
        #ignore the comments between ''' or """
        if line.startswith('"""') or line.startswith("'''"):
            index += comments(index)
            continue
        #identify identifiers
        _identifyClassNames(index+1, line)
        _identifyFunctionNames(index+1, line)
        _identifyVariableNames(index+1, line)
        index += 1
output()
```

本章知识精要

(1) Python 3.x 全面支持中文,Python 2.x 对中文支持还不够,需要在不同的编码格

式之间进行必要的转换。

(2) 对于短字符串，Python 支持驻留机制。

(3) 虽然字符串属于不可变序列，但支持使用 replace()方法以及 maketrans()和 translate()方法进行替换操作，替换时返回新字符串，并不对原字符串做任何修改。

(4) 对用户输入的字符串进行 eval()操作时可能会有安全漏洞。

(5) 可以直接使用 re 模块的方法来进行字符串处理，也可以将模式编译为正则表达式对象，然后使用正则表达式对象的方法来操作字符串。

习　题

1. 假设有一段英文，其中有单独的字母 I 误写为 i，请编写程序进行纠正。

2. 假设有一段英文，其中有单词中间的字母 i 误写为 I，请编写程序进行纠正。

3. 有一段英文文本，其中有单词连续重复了 2 次，编写程序检查重复的单词并只保留一个。例如，文本内容为“This is is a desk.”，程序输出为“This is a desk.”。

4. 简单解释 Python 的字符串驻留机制。

5. 编写程序，用户输入一段英文，然后输出这段英文中所有长度为 3 个字母的单词。

第 5 章　函数设计与使用

在实际开发中，有很多操作是完全相同或者是非常相似的，仅仅是处理的数据不同而已，因此经常会在不同的代码位置多次执行相似或完全相同的代码块。从软件设计和代码复用的角度来讲，很显然，直接将该代码块复制到多个相应的地方然后进行简单修改绝对不是一个好主意。虽然这样可以使得多份复制的代码可以彼此独立地进行修改，但这样不仅增加了代码量，使得程序文件变大，也增加了代码理解和代码维护的难度，更重要的是为代码测试和纠错带来了很大麻烦。一旦被复制的代码块将来某天被发现存在问题而需要修改，则必须将所有的复制都做同样正确的修改，这在实际中是很难完成的一项任务。由于代码量的大幅度增加，导致代码之间的关系更加复杂，很可能在修补了旧漏洞的同时又引入了新漏洞。

解决上述问题的一个常用的方式是设计和编写函数。将可能需要反复执行的代码封装为函数，并在需要执行该段代码功能的地方进行调用，不仅可以实现代码的复用，更重要的是可以保证代码的一致性，只需要修改该函数代码则所有调用位置均受到影响。当然，在实际开发中，需要对函数进行良好的设计和优化才能充分发挥其优势。在编写函数时，有很多原则需要参考和遵守，例如，不要在同一个函数中执行太多的功能，尽量只让其完成一个功能，以提高模块的内聚性。另外，尽量减少不同函数之间的隐式耦合，如减少全局变量的使用，使得函数之间仅通过调用和参数传递来显式体现其相互关系。

在编写函数时，函数体中代码的编写与前面章节介绍的基本一致，只是对代码进行了封装并增加了函数调用、传递参数、返回计算结果等外围接口，这也正是本章讲解的重点。

5.1　函数定义

在 Python 中，定义函数的语法如下：使用 def 关键字来定义函数，然后是一个空格和函数名称，接下来是一对圆括号，在圆括号内是形式参数列表，如果有多个参数则使用逗号分隔开，圆括号之后是一个冒号，最后以换行开始的函数体代码。定义函数时需要注意的问题是：①函数形参不需要声明其类型；②即使该函数不需要接受任何参数，也必须保留一对空的圆括号；③函数体相对于 def 关键字必须保持一定的空格缩进。

```
def 函数名([参数列表]):
    函数体
```

例如，下面的函数用来计算斐波那契数列中小于参数 n 的所有值：

```
def fib(n):
    a, b = 1, 1
    while a < n:
        print(a, end=' ')
```

```
        a, b =b, a+b
    print()
```

该函数的调用方式为

```
fib(1000)
```

5.2 形参与实参

函数定义时圆括弧内是使用逗号分隔开的形参列表，一个函数可以没有形参，但是定义时一对圆括弧必须要有，表示该函数不接受参数。函数调用时向其传递实参，根据不同的参数类型，将实参的值或引用传递给形参。

例如，在 5.1 节中定义函数 fib()时，n 就是该函数的形参，而调用该函数时，1000 则是传递给该函数的实参。

在定义函数时，对参数个数并没有限制，如果有多个形参，则需要使用逗号进行分隔。例如，下面的函数用来接受 2 个参数，并输出其中的最大值。

```
def printMax(a, b):
    if a>b:
        pirnt(a, 'is the max')
    else:
        print(b, 'is the max')
```

当然，这里只是为了演示，而忽略了一些细节，如果输入的参数不支持比较运算，则会出错，可以参考后面第 8 章中介绍的异常处理结构来解决这个问题。

对于绝大多数情况下，在函数内部直接修改形参的值不会影响实参。例如：

```
>>>def addOne(a):
    print(a)
    a +=1
    print(a)
>>>a =3
>>>addOne(a)
3
4
>>>a
3
```

从运行结果可以看出，在函数内部修改了形参 a 的值，但是当函数结束以后，实参 a 的值并没有被修改，可以参考 5.5 节中关于变量作用域的讨论。当然，在有些情况下，可以通过特殊的方式在函数内部修改实参的值，例如：

```
>>>def modify(v):              #修改列表元素值
    v[0] =v[0]+1
>>>a = [2]
>>>modify(a)
```

```
>>>a
[3]
>>>def modify(v, item):        #为列表增加元素
    v.append(item)
>>>a = [2]
>>>modify(a,3)
>>>a
[2, 3]
>>>def modify(d):              #修改字典元素值或为字典增加元素
    d['age'] =38
>>>a ={'name':'Dong', 'age':37, 'sex':'Male'}
>>>a
{'age': 37, 'name': 'Dong', 'sex': 'Male'}
>>>modify(a)
>>>a
{'age': 38, 'name': 'Dong', 'sex': 'Male'}
```

也就是说，如果传递给函数的是 Python 可变序列，并且在函数内部使用下标或其他方式为可变序列增加、删除、修改元素值时，修改后的结果是可以反映到函数之外的。

5.3 参数类型

在 Python 中，函数参数有很多种：可以为普通参数、默认值参数、关键参数、可变长度参数等。Python 函数的定义也非常灵活，在定义函数时不需要指定参数的类型，形参的类型完全由调用者传递的参数类型和 Python 解释器的理解和推断来决定，类似于某些语言中的泛型；同样，也不需要指定函数的返回值类型，这将由函数中的 return 语句来决定。函数的类型由 return 语句返回值的类型来决定，如果函数中没有 return 语句或者没有执行到 return 语句而返回，则函数默认返回空值 None。

由于 Python 程序是解释执行的，因此如果函数编写或设计的有问题，只有在调用时才可能被发现，传递某些类型的参数时执行正确，而传递另一些类型的参数时则可能会出现错误。出现这样的情况有多种可能的原因，例如，不同的参数值可能会使得函数执行不同的路径，或者不同的参数类型所支持的操作和运算符不同，等等。所以，在进行测试时一定要注意，一次或几次运行正常并不表示代码编写的没有问题，必须要进行尽可能完全的测试，尽量满足各种覆盖性要求，尽量在代码发布之前发现和解决更多的潜在问题。

5.3.1 默认值参数

在定义函数时，Python 支持默认值参数，在调用这样的函数时，可以不用为带有默认值的形参进行传值，此时函数将会直接使用函数定义时设置的默认值。语法如下：

```
def 函数名(…,形参名=默认值):
    函数体
```

调用带有默认值参数的函数时，可以不对默认值参数进行赋值，也可以通过显式赋值来替换其默认值，具有较大的灵活性。如果需要的话，可以使用“函数名. func_defaults”(在

Python 3. x 中使用“函数名. __defaults__)随时查看函数所有默认值参数的当前值，返回值为一个元组，其中的元素依次表示每个默认值参数的当前值。例如下面的函数定义：

```
>>>def say( message, times =1 ):
    print ((message+ ' ') * times)[0:-1]
>>>say.func_defaults
(1,)
```

调用该函数时，如果只为第一个参数传递实参，则第二个参数使用默认值 1；如果为第二个参数传递实参，则不再使用默认值 1，而是使用调用者显式传递的值。

```
>>>say('hello')
hello
>>>say('hello',3)
hello hello hello
>>>say('hi',7)
hi hi hi hi hi hi hi
```

再如，下面的函数使用指定分隔符将列表中所有字符串元素连接成一个字符串。

```
>>>def Join(List,sep=None):
    return (sep or ' ').join(List)
>>>aList =['a', 'b', 'c']
>>>Join(aList)
'a b c'
>>>Join(aList, ',')
'a,b,c'
```

需要注意的是，默认值参数必须出现在函数参数列表的最右端，且任何一个默认值参数右边都不能再出现非默认值参数。例如下面的示例，前两个函数不符合这一要求，从而导致函数定义失败，如图 5-1 所示。

```
>>> def f(a=3,b,c=5):
        print a,b,c

SyntaxError: non-default argument follows default argument
>>> def f(a=3,b):
        print a,b

SyntaxError: non-default argument follows default argument
>>> def f(a,b,c=5):
        print a,b,c


>>>
```

图 5-1　带有默认值参数的函数定义

另外，特别需要注意的是，默认值参数只被解释一次，对于复杂类型的默认值参数，这一点可能会导致很严重的逻辑错误，而这种错误或许会耗费您较多的精力来定位和纠正。例如下面的代码：

```
def demo(newitem,old_list=[]):
```

```
        old_list.append(newitem)
        return old_list
    print demo('5',[1,2,3,4])
    print demo('aaa',['a','b'])
    print demo('a')
    print demo('b')
```

您可以试运行一下上面的代码,仔细看看结果,是否能发现问题呢?然后再把代码修改为下面的样子,再试运行一下,看看区别在哪里?然后再仔细阅读本节前面的内容,您应该会发现答案。

```
def demo(newitem,old_list=None):
    if old_list is None:
        old_list = []
    old_list.append(newitem)
    return old_list
print demo('5',[1,2,3,4])
print demo('aaa',['a','b'])
print demo('a')
print demo('b')
```

5.3.2 关键参数

关键参数主要指实参,即调用函数时的参数传递方式,而与函数定义无关。

通过关键参数传递,实参顺序可以和形参顺序不一致,但不影响传递结果,避免了用户需要牢记参数位置与顺序的麻烦,使得函数的调用和参数传递更加灵活方便。

```
>>> def demo(a,b,c=5):
    print a,b,c
>>> demo(3,7)
3 7 5
>>> demo(a=7,b=3,c=6)
7 3 6
>>> demo(c=8,a=9,b=0)
9 0 8
```

5.3.3 可变长度参数

可变长度参数主要有两种形式:* parameter 和**parameter,前者用来接受任意多个实参并将其放在一个元组中,后者接受类似于关键参数一样显式赋值形式的多个实参并将其放入字典中。

下面的代码演示了第一种形式的用法,无论调用该函数时传递了多少实参,一律将其放入元组中:

```
>>> def demo(*p):
    print p
```

```
>>>demo(1,2,3)
(1, 2, 3)
>>>demo(1,2,3,4,5,6,7)
(1, 2, 3, 4, 5, 6, 7)
```

下面的代码则演示了第二种形式的用法，在调用该函数时自动将接受的参数转换为字典：

```
>>>def demo(**p):
    for item in p.items():
        print item
>>>demo(x=1,y=2,z=3)
('y', 2)
('x', 1)
('z', 3)
```

下面的代码演示了几种不同形式的参数混合使用的用法。需要注意的是，虽然 Python 完全支持您这样做，但是除非真的很必要，否则请不要这样用，因为这会使得代码非常混乱而严重降低可读性，并导致程序查错非常困难。

```
>>>def func_4(a,b,c=4,*aa,**bb):
    print (a,b,c)
    print aa
    print bb
>>>func_4(1,2,3,4,5,6,7,8,9,xx='1',yy='2',zz=3)
(1, 2, 3)
(4, 5, 6, 7, 8, 9)
{'yy': '2', 'xx': '1', 'zz': 3}
>>>func_4(1,2,3,4,5,6,7,xx='1',yy='2',zz=3)
(1, 2, 3)
(4, 5, 6, 7)
{'yy': '2', 'xx': '1', 'zz': 3}
```

5.3.4 参数传递的序列解包

传递参数时，可以使用 Python 列表、元组、集合、字典以及其他可迭代对象作为实参，并在实参名称前加一个星号，Python 解释器将自动进行解包，然后传递给多个单变量形参。但需要注意的是，如果使用字典作为实参，则默认使用字典的键，如果需要将字典中的键-值对作为参数则需要使用 items()方法，如果需要将字典的值作为参数则需要调用字典的 values()方法。最后，请保证实参中元素个数与形参个数相等，否则将出现错误。

```
>>>def demo(a,b,c):
    print a+b+c
>>>seq = [1,2,3]
>>>demo(* seq)
6
```

```
>>> tup = (1,2,3)
>>> demo(* tup)
6
>>> dic = {1:'a', 2:'b', 3:'c'}
>>> demo(* dic)
6
>>> Set = {1,2,3}
>>> demo(* Set)
6
```

5.4 return 语句

return 语句用来从一个函数中返回，即结束函数的执行，同时还可用 return 语句从函数中返回一个任意类型的值。不论 return 语句出现在函数的什么位置，一旦得到执行将直接结束函数的执行。如果函数没有 return 语句或者执行了不返回任何值的 return 语句，Python 将认为该函数以 return None 结束，即返回空值。

```
def maximum( x, y ):
    if x> y:
        return x
    else:
        return y
```

作为使用者，在调用函数时，一定要注意函数有没有返回值，是否会对参数的值进行修改。例如，前面介绍过的列表方法 sort()属于原地操作，没有返回值，而内置函数 sorted()则返回排序后的列表，并不对原列表做任何修改。

```
>>> a_list = [1,2,3,4,9,5,7]
>>> print(sorted(a_list))
[1, 2, 3, 4, 5, 7, 9]
>>> print(a_list)
[1, 2, 3, 4, 9, 5, 7]
>>> print(a_list.sort())
None
>>> print(a_list)
[1, 2, 3, 4, 5, 7, 9]
```

5.5 变量作用域

变量起作用的代码范围称为变量的作用域，不同作用域内同名变量之间互不影响。一个变量在函数外部定义和在函数内部定义，其作用域是不同的，函数内部定义的变量一般为局部变量，而不属于任何函数的变量一般为全局变量。一般而言，局部变量的引用比全局变量速度快，应优先考虑使用。除非真的有必要，应尽量避免使用全局变量，因为全局变量会增加不同函数的耦合度，从而降低代码的可读性，并使得代码测试和纠错变得很困难。

在函数内定义的普通变量只在该函数内起作用，称为局部变量。当函数运行结束后，在该函数内部定义的局部变量被自动删除。

如果想要在函数内部修改一个定义在函数外的变量值，那么这个变量就不能是局部的，其作用域必须为全局的，能够同时作用于函数内外，称为全局变量，可以通过 global 来声明或定义。这分两种情况。

(1) 一个变量已在函数外定义，如果在函数内需要修改这个变量的值，并要将这个赋值结果反映到函数外，可以在函数内用 global 声明这个变量，明确声明使用同名的全局变量。

(2) 在函数内部直接将一个变量声明为全局变量，在函数外没有声明，在调用这个函数之后，将增加为新的全局变量。

通过下面的示例代码来演示局部变量和全局变量的用法。

```
>>>def demo():
    global x                #声明或创建全局变量
    x=3                     #修改全局变量的值
    y=4                     #局部变量
    print x,y
>>>x=5
>>>demo()                   #本次调用修改了全局变量 x 的值
3 4
>>>x
3
>>>y                        #局部变量在函数运行结束之后自动删除
出错信息
NameError: name 'y' is not defined
>>>del x
>>>x
出错信息
NameError: name 'x' is not defined
>>>demo()                   #本次调用创建了全局变量
3 4
>>>x
3
>>>y
出错信息
NameError: name 'y' is not defined
```

5.6 lambda 表达式

lambda 表达式可以用来声明匿名函数，即没有函数名字的临时使用的小函数。lambda 表达式只可以包含一个表达式，不允许包含其他复杂的语句，但在表达式中可以调用其他函数，该表达式的计算结果是函数的返回值。下面的代码演示了不同情况下 lambda 表达式的应用。

```
>>> f = lambda x,y,z:x+ y+ z
>>> print f(1,2,3)
6
>>> g = lambda x,y= 2,z= 3:x+ y+ z
>>> print g(1)
6
>>> print g(2, z= 4, y= 5)
11
>>> L= [(lambda x:x**2),(lambda x:x**3),(lambda x:x**4)]
>>> print L[0](2), L[1](2), L[2](2)
4 8 16
>>> D= {'f1':(lambda:2+ 3),'f2':(lambda:2 * 3),'f3':(lambda:2**3)}
>>> print D['f1'](), D['f2'](), D['f3']()
5 6 8
>>> L= [1,2,3,4,5]
>>> print map((lambda x:x+ 10), L)
[11, 12, 13, 14, 15]
>>> L
[1, 2, 3, 4, 5]
>>> def demo(n):
    return n * n
>>> demo(5)
25
>>> a_list = [1,2,3,4,5]
>>> map(lambda x:demo(x), a_list)
[1, 4, 9, 16, 25]
>>> data = list(range(20))
>>> print data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> import random
>>> random.shuffle(data)
>>> data
[4, 3, 11, 13, 12, 15, 9, 2, 10, 6, 19, 18, 14, 8, 0, 7, 5, 17, 1, 16]
>>> data.sort(key= lambda x:x)      #用在列表的 sort()方法中
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> data.sort(key= lambda x:len(str(x)))
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> data.sort(key= lambda x:len(str(x)), reverse= True)
>>> data
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

5.7 高级话题

在本章的最后，我们来看几个高级一点的话题，包括 map()、reduce()、filter()、生成器以及 Python 字节码的知识。

(1) 内置函数 map()可以将一个单参数函数依次作用到一个序列或迭代器对象的每个元素上，并返回一个列表作为结果，该列表中的每个元素是原序列中的元素经过函数处理后的结果，该函数不对原序列或迭代器对象做任何修改。

```
>>>map(str,range(5))
['0', '1', '2', '3', '4']
>>>def add5(v):
    return v+5
>>>map(add5,range(10))
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

(2) 内置函数 reduce()可以将一个接受 2 个参数的函数以累积的方式从左到右依次作用到一个序列或迭代器对象的所有元素上。

```
>>>seq=[1,2,3,4,5,6,7,8,9]
>>>reduce(lambda x,y:x+y, seq)
45
>>>def add(x, y):
    return x + y
>>>reduce(add,range(10))
45
```

上面的代码运行过程如图 5-2 所示。

类似的运算并不局限于数值类型，例如，下面的代码使用前面定义的函数 add()实现了字符串连接。

```
>>>reduce(add,map(str,range(10)))
'0123456789'
```

注：在 Python 3 中，使用 reduce()函数需要先从 functools 模块导入，即

```
from functools import reduce
```

(3) 内置函数 filter()将一个单参数函数作用到一个序列上，返回该序列中使得该函数返回值为 True 的那些元素组成的列表、元组或字符串。

```
>>>seq=['foo','x41','?!','***']
>>>def func(x):
    return x.isalnum()
>>>filter(func,seq)
['foo', 'x41']
>>>seq
```

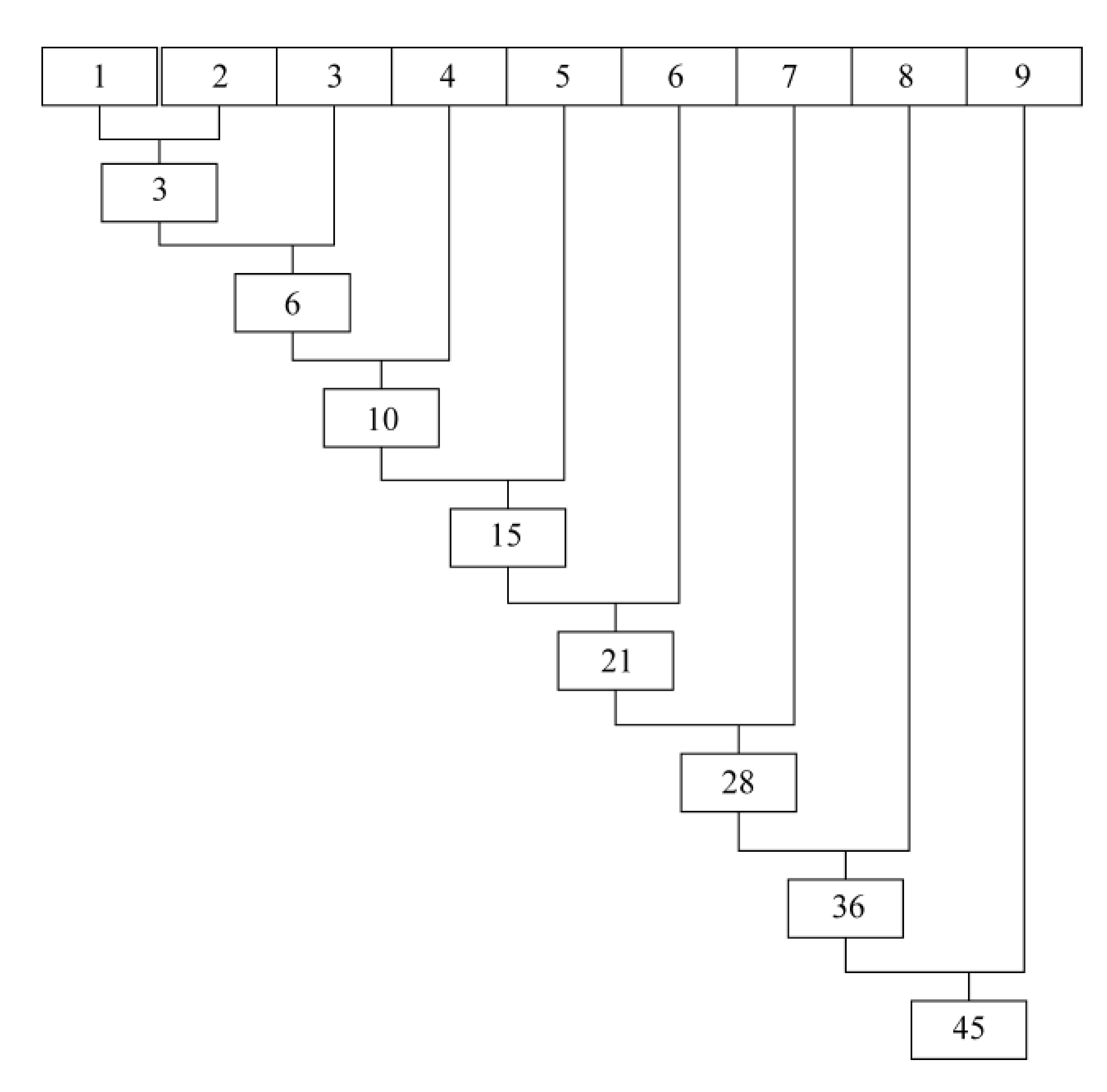


图 5-2　reduce 过程示意图

```
['foo', 'x41', '?!', '* * * ']
>>> [x for x in seq if x.isalnum()]
['foo', 'x41']
>>> filter(lambda x:x.isalnum(),seq)
['foo', 'x41']
```

（4）包含 yield 语句的函数用来创建生成器。迭代器的最大特点是惰性求值，它尤其适用于大数据处理。

```
>>> def f():
    a, b = 1, 1
    while True:
        yield a
        a, b = b, a+b
>>> a = f()
>>> for i in range(10):
    print(a.__next__(), end=' ')
1 1 2 3 5 8 13 21 34 55
```

上面定义的生成器函数还可以这样使用：

```
>>> for i in f():
    if i > 100:
        break
    print(i, end=' ')
1 1 2 3 5 8 13 21 34 55 89
```

（5）使用 dis 模块的功能可以查看函数的字节码指令。

```
>>>def add(n):
    n+=1
    return n
>>>import dis
>>>dis.dis(add)
2       0 LOAD_FAST          0 (n)
        3 LOAD_CONST         1 (1)
        6 INPLACE_ADD
        7 STORE_FAST         0 (n)

3       10 LOAD_FAST         0 (n)
        13 RETURN_VALUE
```

本章知识精要

（1）函数是用来实现代码复用的常用方式。

（2）定义函数时使用关键字 def。

（3）定义函数时不需要指定其形参类型，而是根据调用函数时传递的实参自动进行推断。

（4）定义函数时可以为形参设置默认值，如果调用该函数时不为默认值参数传递参数，将自动使用默认值。

（5）传递参数时可以使用关键参数，避免牢记参数顺序的麻烦。

（6）定义函数时，形参前面加一个星号表示可以接收多个实参并将其放置到一个元组中，形参前面加两个星号表示可以接收多个“键-值对”参数并将其放置到字典中。

（7）调用函数时，在实参前面加一个星号表示进行序列解包，可以将序列中的值依次赋值给相应数量的形参。

（8）定义函数时不需要指定其返回值的类型，而是由 return 语句来决定，如果函数中没有 return 语句或执行了不返回任何值的 return 语句，则 Python 认为该函数返回空值 None。

习　　题

1. 运行 5.3.1 节最后的示例代码，查看结果并分析原因。
2. 编写函数，判断一个整数是否为素数，并编写主程序调用该函数。
3. 编写函数，接收一个字符串，分别统计大写字母、小写字母、数字、其他字符的个数，并以元组的形式返回结果。
4. 在 Python 程序中，局部变量会隐藏同名的全局变量吗？请编写代码进行验证。
5. 编写函数，可以接收任意多个整数并输出其中的最大值和所有整数之和。
6. 编写函数，模拟内置函数 sum()。
7. 编写函数，模拟内置函数 sorted()。

第 6 章　面向对象程序设计

面向对象程序设计(Object Oriented Programming,OOP)的思想主要针对大型软件设计而提出,使得软件设计更加灵活,能够很好地支持代码复用和设计复用,并且使得代码具有更好的可读性和可扩展性。面向对象程序设计的一条基本原则是计算机程序由多个能够起到子程序作用的单元或对象组合而成,这大大地降低了软件开发的难度,使得编程就像搭积木一样简单。面向对象程序设计的一个关键性观念是将数据以及对数据的操作封装在一起,组成一个相互依存、不可分割的整体,即对象。对于相同类型的对象进行分类、抽象后,得出共同的特征而形成了类,面向对象程序设计的关键就是如何合理地定义和管理这些类以及类之间的关系。

Python 完全采用了面向对象程序设计的思想,是真正面向对象的脚本语言,完全支持面向对象的基本功能,如封装、继承、多态以及对基类方法的覆盖或重写。但与其他面向对象程序设计语言不同的是,Python 中对象的概念很广泛,Python 中的一切内容都可以称为对象,而不一定必须是某个类的实例。例如,字符串、列表、字典、元组等内置数据类型都具有和类完全相似的语法和用法。创建类时用变量形式表示的对象属性称为数据成员或成员属性,用函数形式表示的对象行为称为成员函数或成员方法,成员属性和成员方法统称为类的成员。

6.1　类的定义与使用

Python 使用 class 关键字来定义类,class 保留字之后是一个空格,然后是类的名字,再然后是一个冒号,最后换行并定义类的内部实现。类名的首字母一般要大写,当然也可以按照自己的习惯定义类名,但是一般推荐参考惯例来命名,并在整个系统的设计和实现中保持风格一致,这一点对于团队合作尤其重要。例如:

```
class Car:                    #新式类必须有至少一个基类
    def infor(self):
        print(" This is a car ")
```

类的所有实例方法都必须至少有一个名为 self 的参数,并且必须是方法的第一个形参(如果有多个形参的话),self 参数代表将来要创建的对象本身。在类的实例方法中访问实例属性时需要以 self 为前缀,但调用对象方法时并不需要传递这个参数。

注:在定义类的方法时,一般习惯将第一个参数定义为 self,但实际上类的实例方法中第一个参数的名字是可以变化的,而不必须使用 self 这个名字,例如:

```
>>>class A:
    def __init__(hahaha,v):
        hahaha.value =v
```

```
    def show(hahaha):
        print hahaha.value
>>> a = A(3)
>>> a.show()
3
```

属性有两种：一种是实例属性；另一种是类属性。实例属性一般是指在构造函数__init__()中定义的，定义时以 self 作为前缀；类属性是在类中所有方法之外定义的数据成员。在主程序中（或类的外部），实例属性属于实例（对象），只能通过对象名访问，而类属性属于类，可以通过类名或对象名访问。

在类的方法中可以调用类本身的其他方法，也可以访问类属性以及对象属性。在 Python 中比较特殊的是，可以动态地为类和对象增加成员，这一点是和很多面向对象程序设计语言不同的，也是 Python 动态类型特点的一种重要体现。

```
class Car:
    price = 100000                  #定义类属性
    def __init__(self, c):
        self.color = c              #定义实例属性
car1 = Car("Red")
car2 = Car("Blue")
print car1.color, Car.price
Car.price = 110000                  #修改类属性
Car.name = 'QQ'                     #增加类属性
car1.color = "Yellow"               #修改实例属性
print car2.color, Car.price, Car.name
print car1.color, Car.price, Car.name
```

Python 并没有对私有成员提供严格的访问保护机制。在定义类的属性时，如果属性名以两个下划线“__”开头则是私有属性，否则是公有属性。私有属性在类外不能直接访问，需要通过调用对象的公有成员方法来访问，或者通过 Python 支持的特殊方式来访问。Python 提供了访问私有属性的特殊方式，可用于程序的测试和调试，对于成员方法也具有同样的性质。

私有属性是为了数据封装和保密而设的属性，一般只能在类的成员方法（类的内部）中使用访问，虽然 Python 支持一种特殊的方式来从外部直接访问类的私有成员，但是并不推荐读者这样做。公有属性是可以公开使用的，既可以在类的内部进行访问，也可以在外部程序中使用。

```
>>> class A:
    def __init__(self, value1 = 0, value2 = 0):
        self._value1 = value1
        self.__value2 = value2
    def setValue(self, value1, value2):
        self._value1 = value1
        self.__value2 = value2
    def show(self):
```

```
        print self._value1
        print self.__value2
>>>a =A()
>>>a._value1
0
>>>a._A__value2        #在外部访问对象的私有数据成员
0
```

在 IDLE 环境中，在对象或类名后面加上一个圆点“.”，稍等一会则会自动列出其所有公开成员，例如图 6-1 所示，模块也具有同样的特点。

如果在圆点“.”后面再加一个下划线，则会列出该对象或类的所有成员，包括私有成员，如图 6-2 所示。

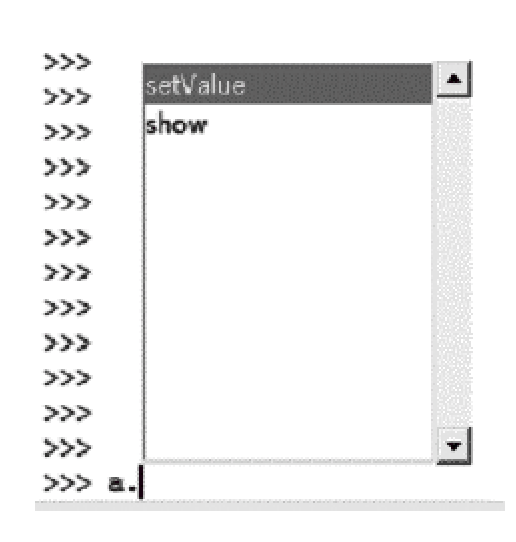




图 6-1 列出对象公开成员

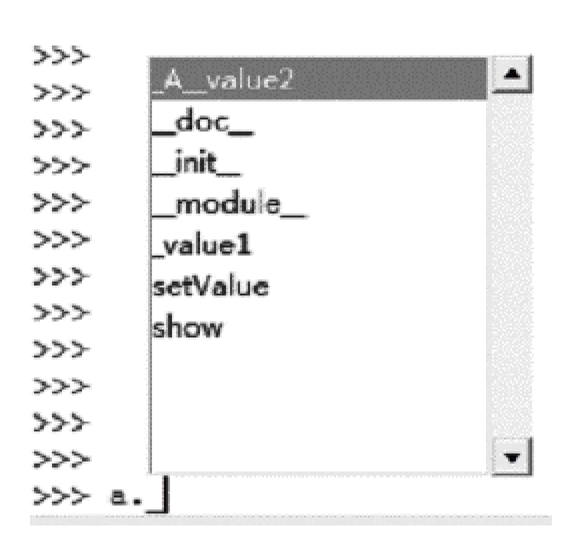


图 6-2 列出对象所有成员

在 Python 中，以下划线开头的变量名有特殊的含义，尤其是在类和模块的定义中。用下划线作为变量前缀和后缀来表示类的特殊成员。

(1) _xxx：这样的对象称为保护变量，不能用'from module import *'导入，只有类对象和子类对象能访问这些变量。

(2) __xxx__：系统定义的特殊成员。

(3) __xxx：类中的私有成员，只有类对象自己能访问，子类对象也不能访问这个成员，但在对象外部可以通过“对象名._类名__xxx”这样的特殊方式来访问。Python 中不存在真正意义上的私有成员。

另外，在 IDLE 交互模式下，一个下划线“_”表示解释器中最后一次显示的内容或最后一次语句正确执行的输出结果。例如：

```
>>>3 +5
8
>>>_ +2
10
>>>_ * 3
30
>>>_ / 5
6
>>>3
3
>>>_
```

```
3
>>>1/0

Traceback (most recent call last):
    File "<pyshell#2>", line 1, in <module>
        1/0
ZeroDivisionError: integer division or modulo by zero
>>>_
3
```

下面的代码演示了特殊成员定义和访问的方法。

```
>>>class Fruit:
    def __init__(self):
        self.__color = 'Red'
        self.price = 1
>>>apple = Fruit()
>>>apple.price                              #显示对象公开数据成员的值
1
>>>apple.price = 2                          #修改对象公开数据成员的值
>>>apple.price
2
>>>print apple.price, apple._Fruit__color   #显示对象私有数据成员的值
2 Red
>>>apple._Fruit__color = "Blue"             #修改对象私有数据成员的值
>>>print apple.price, apple._Fruit__color
2 Blue
>>>print apple.__color                      #不能直接访问对象的私有数据成员,出错
Traceback (most recent call last):
    File "<pyshell#16>", line 1, in <module>
        print(apple.__color )
AttributeError: Fruit instance has no attribute '__color'
>>>peach = Fruit()
>>>print peach.price, peach._Fruit__color
1 Red
```

6.2 类的方法

类的方法可以粗略分为四大类：公有方法、私有方法、静态方法和类方法。其中，公有方法、私有方法都属于对象，每个对象都有自己的公有方法和私有方法，在这两类方法中可以访问属于类和对象的成员；公有方法通过对象名直接调用，私有方法不能通过对象名直接调用，只能在属于对象的方法中通过 self 调用或在外部通过 Python 支持的特殊方式来调用；静态方法和类方法都可以通过类名和对象名调用，但不能访问属于对象的成员，只能访问属于类的成员。

```
>>>class Root:
    __total = 0
    def __init__(self,v):
        self.__value = v
        Root.__total += 1
    def show(self):
        print 'self.__value:',self.__value
        print 'Root.__total:',Root.__total
    @classmethod
    def classShowTotal(cls):                    #类方法
        print cls.__total
    @staticmethod
    def staticShowTotal():                      #静态方法
        print Root.__total
>>>r = Root(3)
>>>r.classShowTotal()                           #通过对象来调用类方法
1
>>>r.staticShowTotal()                          #通过对象来调用静态方法
1
>>>r.show()
self.__value: 3
Root.__total: 1
>>>rr = Root(3)
>>>Root.classShowTotal()                        #通过类名调用类方法
2
>>>Root.staticShowTotal()                       #通过类名调用静态方法
2
>>>Root.show()                                  #试图通过类名直接调用实例方法,失败
Traceback (most recent call last):
    File "<pyshell#9>", line 1, in <module>
        Root.show()
TypeError: unbound method show() must be called with Root instance as first argument (got
nothing instead)
>>>Root.show(r)                                #可以通过这种方法来调用方法并访问实例成员
self.__value: 3
Root.__total: 2
>>>r.show()
self.__value: 3
Root.__total: 2
```

6.3 类的属性

Python 2.x 和 Python 3.x 对属性的处理方式不一样,有较大的差异,使用时应注意两者之间的区别。需要注意的是,本节中“属性”指狭义的概念,与前面所谈的“属性”概念并不完全一样。

6.3.1 Python 2.x 中的属性

在 Python 2 中，使用@property 来声明一个属性，然而属性并没有得到真正的实现，也没有提供应有的访问保护机制。正如前面所说，在 Python 中，可以为类和对象动态增加新成员。在 Python 2 中，为对象增加新的数据成员时，将隐藏同名的已有属性。例如下面的 Python 2.7.8 代码：

```
>>>class Test:
    def __init__(self, value):
        self.__value = value
    @property
        def value(self):
            return self.__value
>>>a = Test(3)
>>>a.value
3
>>>a.value = 5                  #动态添加新成员,隐藏了定义的属性
>>>a.value
5
>>>t._Test__value               #原来的私有变量没有改变
3
```

除了动态增加成员时会隐藏已有属性，下面的代码从表面看来是修改属性的值，而实际上也是增加了新成员，从而隐藏了已有属性。

```
>>>class Test:
    def __init__(self, value):
        self.__value = value
    def __get(self):
        return self.__value
    def __set(self, v):
        self.__value = v
    value = property(__get, __set)
    def show(self):
        print self.__value
>>>t = Test(3)
>>>t.value
3
>>>t.value += 2                 #动态添加了新成员
>>>t.value                      #这里访问的是新成员
5
>>>t.show()                     #访问原来定义的私有数据成员
3
>>>del t.value                  #这里删除的是刚才添加的新成员
>>>t.value                      #访问原来的属性
```

```
3
>>>del t.value                  #试图删除属性
出错信息(略)
AttributeError: Test instance has no attribute 'value'
>>>del t._Test__value           #删除私有成员
>>>t.value                      #访问属性,但该属性对应的私有成员已不存在
出错信息(略)
AttributeError: Test instance has no attribute '_Test__value'
```

下面的代码则更加清楚地演示了 Python 2 中私有成员和普通成员之间的关系,希望能帮助您理解上面的内容。

```
>>>class Test:
    def show(self):
        print self.value
        print self.__v
>>>t =Test()
>>>t.show()
出错信息(略)
AttributeError: Test instance has no attribute 'value'
>>>t.value =3
>>>t.show()
3
出错信息(略)
AttributeError: Test instance has no attribute '_Test__v'
>>>t.__v =5
>>>t.show()
3
出错信息(略)
AttributeError: Test instance has no attribute '_Test__v'
>>>t._Test__v =5
>>>t.show()
3
5
```

6.3.2 Python 3.x 中的属性

在 Python 3 中,属性得到了较为完整的实现,支持更加全面的保护机制。如下面的代码所示,如果设置属性为只读,则无法修改其值,也无法为对象增加与属性同名的新成员,同时,也无法删除对象属性。例如,下面的代码运行在 Python 3.4.2 中:

```
>>>class Test:
    def __init__(self, value):
        self.__value =value
    @property
    def value(self):            #只读,无法修改和删除
```

```
        return self.__value
>>>t = Test(3)
>>>t.value
3
>>>t.value = 5
出错信息 (略)
AttributeError: can't set attribute
>>>t.v=5                         #动态增加新成员
>>>t.v
5
>>>del t.v                       #动态删除成员
>>>del t.value                   #试图删除对象属性,失败
出错信息 (略)
AttributeError: can't delete attribute
>>>t.value
3
```

下面的代码则把属性设置为可读、可修改,而不允许删除。

```
>>>class Test:
    def __init__(self, value):
        self.__value = value
    def __get(self):
        return self.__value
    def __set(self, v):
        self.__value = v
    value = property(__get, __set)
>>>t = Test(3)
>>>t.value                       #允许读取属性值
3
>>>t.value = 5                   #允许修改属性值
>>>t.value
5
>>>del t.value                   #试图删除属性,失败
出错信息
AttributeError: can't delete attribute
```

当然,也可以将属性设置为可读、可修改、可删除。

```
>>>class Test:
    def __init__(self, value):
        self.__value = value
    def __get(self):
        return self.__value
    def __set(self, v):
        self.__value = v
    def __del(self):
```

```
        del self.__value
    value = property(__get, __set, __del)
    def show(self):
        print(self.__value)
>>> t = Test(3)
>>> t.show()
3
>>> t.value
3
>>> t.value = 5
>>> t.show()
5
>>> t.value
5
>>> del t.value
>>> t.value
出错信息(略)
AttributeError: 'Test' object has no attribute '_Test__value'
>>> t.show()
出错信息(略)
AttributeError: 'Test' object has no attribute '_Test__value'
>>> t.value = 1                #为对象动态增加属性和对应的私有数据成员
>>> t.show()
1
>>> t.value
1
```

6.4 类的特殊方法

Python 类有大量的特殊方法,比较常见的是构造函数和析构函数。Python 中类的构造函数是__init__(),一般用来为数据成员设置初值或进行其他必要的初始化工作,在创建对象时自动执行。如果用户没有设计构造函数,Python 将提供一个默认的构造函数用来进行必要的初始化工作。Python 中类的析构函数是__del__(),一般用来释放对象占用的资源,在 Python 删除对象和收回对象空间时自动执行。如果用户没有编写析构函数,Python 将提供一个默认的析构函数进行必要的清理工作。

在 Python 中,除了构造函数和析构函数之外,还有大量的特殊方法支持更多的功能,例如运算符重载就是通过在类中重写特殊函数来实现的。表 6-1 列出了其中一部分特殊方法。

表 6-1 Python 类特殊方法

方　　法	功 能 说 明
__init__()	构造函数,生成对象时调用
__del__()	析构函数,释放对象时调用

续表

方　　法	功能说明
__add__()、__radd__()	左+、右+
__sub__()	-
__mul__()	*
__div__()、__truediv__()	Python 2 使用__div__()，Python 3 使用__truediv__()
__floordiv__()	整除
__mod__()	%
__pow__()	**
__cmp__()	比较运算
__repr__()	打印、转换
__setitem__()	按照索引赋值
__getitem__()	按照索引获取值
__len__()	计算长度
__call__()	函数调用
__contains__ ()	测试是否包含某个元素
__eq__()、__ne__()、__lt__()、__le__()、__gt__()、__ge__()	==、!=、<、<=、>、>=
__str__()	转化为字符串
__lshift__()、__rshift__()	<<、>>
__and__()、__or__()、__invert__()	&、|、~
__iadd__()、__isub__()	+=、-=

下面通过一个示例来演示特殊方法的用法。

【例 6-1】 自定义数组类。

```
#Filename: MyArray.py
#Function description: Array and its operating
#--------------------

import types

class MyArray:
    "All the elements in this array must be numbers"
    __value = []
    __size = 0

    def __IsNumber(self, n):
        if type(n) != types.ComplexType and type(n) != types.FloatType \
```

```
                and type(n) !=types.IntType and type(n) !=types.LongType:
            return False
        return True

    def __init__(self, * args):
        for arg in args:
            if not self.__IsNumber(arg):
                print 'All elements must be numbers'
                return
        self.__value = []
        for arg in args:
            self.__value.append(arg)
        self.__size =len(args)

    def __add__(self, n):
        if not self.__IsNumber(n):
            print '+operating with ', type(n), ' and number type is not supported.'
            return
        b =MyArray()
        for v in self.__value:
            b.__value.append(v +n)
        return b

    def __sub__(self, n):
        if not self.__IsNumber(n):
            print '-operating with ', type(n), ' and number type is not supported.'
            return
        b =MyArray()
        for v in self.__value:
            b.__value.append(v -n)
        return b

    def __mul__(self, n):
        if not self.__IsNumber(n):
            print '* operating with ', type(n), ' and number type is not supported.'
            return
        b =MyArray()
        for v in self.__value:
            b.__value.append(v * n)
        return b

    def __div__(self, n):
        if not self.__IsNumber(n):
            print r'/ operating with ', type(n), ' and number type is not supported.'
```

```
            return
        if type(n) ==types.IntType:
            n =float(n)
        b =MyArray()
        for v in self.__value:
            b.__value.append(v / n)
        return b

    def __mod__(self, n):
        if not self.__IsNumber(n):
            print r'%operating with ', type(n), ' and number type is not supported.'
            return
        b =MyArray()
        for v in self.__value:
            b.__value.append(v %n)
        return b

    def __pow__(self, n):
        if not self.__IsNumber(n):
            print '** operating with ', type(n), ' and number type is not supported.'
            return
        b =MyArray()
        for v in self.__value:
            b.__value.append(v * * n)
        return b

    def __len__(self):
        return len(self.__value)

    # for: x
    # when use the object as a statement directly, the function will be called
    def __repr__(self):
        # equivalent to return 'self.__value'
        return repr(self.__value)

    # for: print x
    def __str__(self):
        return str(self.__value)

    def append(self, v):
        if not self.__IsNumber(v):
            print 'Only number can be appended.'
            return
        self.__value.append(v)
```

```
        self.__size +=1

    def __getitem__(self, index):
        if self.__IsNumber(index) and 0 <=index <=self.__size:
            return self.__value[index]
        else:
            print 'Index out of range.'

    def __setitem__(self, index, v):
        if self.__IsNumber(index) and 0 <=index <=self.__size:
            if self.__IsNumber(v):
                self.__value[index] =v
            else:
                print v, ' is not a number'
        else:
            print index, ' is not a number or out of range.'

    #member test. support the keyword 'in'
    def __contains__(self, v):
        if v in self.__value:
            return True
        return False

    #dot product
    def dot(self, v):
        if not isinstance(v, MyArray):
            print v, ' must be an instance of MyArray.'
            return
        if len(v) !=self.__size:
            print 'The size must be equal.'
            return
        b =MyArray()
        for m, n in zip(v.__value, self.__value):
            b.__value.append(m * n)
        return sum(b.__value)

    #equal to
    def __eq__(self, v):
        if not isinstance(v, MyArray):
            print v, ' must be an instance of MyArray.'
            return
        if cmp(self.__value, v.__value) ==0:
            return True
        return False
```

```
    # less than
    def __lt__(self, v):
        if not isinstance(v, MyArray):
            print v, ' must be an instance of MyArray.'
            return
        if cmp(self.__value, v.__value) < 0:
            return True
        return False

if __name__ == '__main__':
    print 'Please use me as a module.'
```

将上面的代码保存为 MyArray.py 文件之后，将其作为模块导入来使用，简单演示如下：

```
>>> import MyArray
>>> a = MyArray.MyArray(1,2,3,4,5,6)
>>> b = MyArray.MyArray(6,5,4,3,2,1)
>>> len(a)
6
>>> a.dot(b)
56
>>> a < b
True
>>> a > b
False
>>> a == a
True
>>> 3 in a
True
>>> a * 3
[3, 6, 9, 12, 15, 18]
>>> a + 2
[3, 4, 5, 6, 7, 8]
>>> a ** 2
[1, 4, 9, 16, 25, 36]
>>> a / 2
[0.5, 1.0, 1.5, 2.0, 2.5, 3.0]
>>> a
[1, 2, 3, 4, 5, 6]
>>> a[0] = 8
>>> a
[8, 2, 3, 4, 5, 6]
```

6.5 继 承 机 制

继承是为代码复用和设计复用而设计的,是面向对象程序设计的重要特性之一。当设计一个新类时,如果可以继承一个已有的设计良好的类然后进行二次开发,无疑会大幅度减少开发工作量。在继承关系中,已有的、设计好的类称为父类或基类,新设计的类称为子类或派生类。派生类可以继承父类的公有成员,但是不能继承其私有成员。

Python 支持多继承,如果父类中有相同的方法名,而在子类中使用时没有指定父类名,则 Python 解释器将从左向右按顺序进行搜索。

【例 6-2】 设计 Person 类,并根据 Person 类派生 Teacher 类,分别创建 Person 类与 Teacher 类的对象。

```
import types
class Person(object):          #基类必须继承于 object,否则在派生类中将无法使用 super()函数
    def __init__(self, name = '', age = 20, sex = 'man'):
        self.setName(name)
        self.setAge(age)
        self.setSex(sex)
    def setName(self, name):
        if type(name) != types.StringType:
            print 'name must be string.'
            return
        self.__name = name
    def setAge(self, age):
        if type(age) != types.IntType:
            print 'age must be integer.'
            return
        self.__age = age
    def setSex(self, sex):
        if sex != 'man' and sex != 'woman':
            print 'sex must be "man" or "woman"'
            return
        self.__sex = sex
    def show(self):
        print self.__name
        print self.__age
        print self.__sex
class Teacher(Person):
    def __init__(self, name='', age = 30, sex = 'man', department = 'Computer'):
        #调用基类构造方法初始化基类的私有数据成员
        super(Teacher, self).__init__(name, age, sex)
        #Person.__init__(self, name, age, sex)        #也可以这样初始化基类的私有数据成员
        self.setDepartment(department)                #初始化派生类的数据成员
    def setDepartment(self, department):
```

```
            if type(department) !=types.StringType:
                print 'department must be a string.'
                return
            self.__department =department
        def show(self):
            super(Teacher, self).show()
            print self.__department
if __name__ =='__main__':
    zhangsan =Person('Zhang San', 19, 'man')
    zhangsan.show()
    lisi =Teacher('Li Si',32, 'man', 'Math')
    lisi.show()
```

为了更好地理解 Python 类的继承机制，我们来看下面的代码，请认真体会构造函数、私有方法以及普通公开方法的继承原理。

```
>>>class A():
    def __init__(self):
        self.__private()
        self.public()
    def __private(self):
        print '__private() method of A'
    def public(self):
        print 'public() method of A'
>>>class B(A):
    def __private(self):
        print '__private() method of B'
    def public(self):
        print 'public() method of B'
>>>b=B()
__private() method of A
public() method of B
>>>print '\n'.join(dir(b))                          #查看对象 b 的成员
_A__private
_B__private
__doc__
__init__
__module__
Public
>>>class C(A):
    def __init__(self):
        self.__private()
        self.public()
    def __private(self):
        print '__private() method of C'
    def public(self):
```

```
        print 'public() method of C'
>>> c=C()
__private() method of C
public() method of C
>>> print '\n'.join(dir(c))
_A__private
_C__private
__doc__
__init__
__module__
public
```

本章知识精要

(1) 定义类时使用关键字 class。

(2) Python 2.x 中的属性并没有提供完整的保护机制。

(3) 在 Python 中,运算符重载是通过重新实现一些特殊函数来实现的。

(4) Python 支持多继承,如果父类中有相同的方法名,而在子类中使用时没有指定父类名,则 Python 解释器将从左向右按顺序进行搜索。

习　题

1. 继承 6.5 节例 6-2 中的 Person 类生成 Student 类,填写新的函数用来设置学生专业,然后生成该类对象并显示信息。

2. 设计一个三维向量类,并实现向量的加法、减法以及向量与标量的乘法和除法运算。

3. 面向对象程序设计的三要素分别为________、________和________。

4. 简单解释 Python 中以下划线开头的变量名特点。

5. 与运算符 ** 对应的特殊方法名为________,与运算符 // 对应的特殊方法名为________。

第7章　文件操作

为了长期保存数据以便重复使用、修改和共享，必须将数据以文件的形式存储到外部存储介质（如磁盘、U 盘、光盘等）或云盘中。管理信息系统是使用数据库来存储数据的，而数据库最终还是要以文件的形式存储到硬盘或其他存储介质上，应用程序的配置信息往往是使用文件来存储的，图形、图像、音频、视频、可执行文件等也都是以文件的形式存储在磁盘上的。因此，文件操作在各类软件的开发中均占有重要的地位。

按文件中数据的组织形式可以把文件分为文本文件和二进制文件两大类。

1. 文本文件

文本文件存储的是常规字符串，由若干文本行组成，通常每行以换行符\n 结尾。常规字符串是指记事本或其他文本编辑器能正常显示、编辑并且人类能够直接阅读和理解的字符串，如英文字母、汉字、数字字符串。文本文件可以使用字处理软件（如 gedit、记事本）进行编辑。

2. 二进制文件

二进制文件把对象内容以字节串（bytes）进行存储，无法用记事本或其他普通字处理软件直接进行编辑，通常也无法被人类直接阅读和理解，需要使用专门的软件进行解码后读取、显示、修改或执行。常见的如图形图像文件、音视频文件、可执行文件、资源文件、各种数据库文件、各类 Office 文档等都属于二进制文件。

7.1　文件基本操作

无论是文本文件还是二进制文件，其操作流程基本都是一致的，即：首先打开文件并创建文件对象，然后通过该文件对象对文件内容进行读取、写入、删除、修改等操作，最后关闭并保存文件内容。Python 内置了文件对象，通过 open()函数可以按指定模式打开指定文件并创建文件对象，例如：

```
文件对象名 =open(文件名[, 打开方式[, 缓冲区]])
```

其中，文件名指定了被打开的文件名称，如果要打开的文件不在当前目录中，还需要指定完整路径，为了减少完整路径中\符号的输入，可以使用原始字符串；打开模式（见表 7-1）指定了打开文件后的处理方式，例如“只读”、“读写”、“追加”等；缓冲区指定了读写文件的缓存模式，数值 0 表示不缓存，数值 1 表示缓存，如大于 1 则表示缓冲区的大小，默认值是缓存模式。如果执行正常，open()函数返回 1 个文件对象，通过该文件对象可以对文件进行各种操作，如果指定文件不存在、访问权限不够、磁盘空间不够或其他原因导致创建文件对象失败则抛出异常。例如，下面的代码分别以读、写方式打开了两个文件并创建了与之对应的文件对象。

```
f1=open( 'file1.txt', 'r' )
```

```
f2=open('file2.txt', 'w')
```

当对文件内容操作完以后，一定要关闭文件，以保证所做的任何修改都得到保存。

```
f1.close()
```

表 7-1　文件打开模式

模式	说　明	模式	说　明
r	读模式	b	二进制模式(可与其他模式组合使用)
w	写模式	+	读、写模式(可与其他模式组合使用)
a	追加模式		

文件对象常用属性如表 7-2 所示。

表 7-2　文件对象常用属性

属　性	说　明
Closed	判断文件是否关闭，若文件被关闭，则返回 True
Mode	返回文件的打开模式
Name	返回文件的名称

文件对象常用方法如表 7-3 所示。

表 7-3　文件对象常用方法

方　法	功能说明
flush()	把缓冲区的内容写入文件，但不关闭文件
close()	把缓冲区的内容写入文件，同时关闭文件，并释放文件对象
read([size])	从文件中读取 size 个字节(Python 2.x)或字符(Python 3.x)的内容作为结果返回，如果省略 size 则表示一次性读取所有内容
readline()	从文本文件中读取一行内容作为结果返回
readlines()	把文本文件中的每行文本作为一个字符串存入列表中，返回该列表
seek(offset[,whence])	把文件指针移动到新的位置，offset 表示相对于 whence 的位置。whence 为 0 表示从文件头开始计算，Whence 为 1 表示从当前位置开始计算，Whence 为 2 表示从文件尾开始计算，默认为 0
tell()	返回文件指针的当前位置
truncate([size])	删除从当前指针位置到文件末尾的内容。如果指定了 size，则不论指针在什么位置都只留下前 size 个字节，其余的删除
write(s)	把字符串 s 的内容写入文件
writelines(s)	把字符串列表写入文本文件，不添加换行符

7.2 文本文件基本操作

在本节中，主要通过几个示例来演示文本文件的读写操作。对于 read()、write()以及其他读写方法，当读写操作完成之后，都会自动移动文件指针，如果需要对文件指针进行定位，可以使用 seek()方法，如果需要获知文件指针当前位置可以使用 tell()方法。

【例 7-1】 向文本文件中写入内容。

```
f = open('sample.txt', 'a+')
s = '文本文件的读取方法\n 文本文件的写入方法\n'
f.write(s)
f.close()
```

对于上面的代码，更建议这样写：

```
s = '文本文件的读取方法\n 文本文件的写入方法\n'
with open('sample.txt','a+') as f:
    f.write(s)
```

使用上下文管理关键字 with 可以自动管理资源，不论何种原因跳出 with 块，总能保证文件被正确关闭，并且可以在代码块执行完毕后自动还原进入该代码块时的现场。

【例 7-2】 读取并显示文本文件的前 5 个字节。

对于文件对象的 read()方法，Python 2 和 Python 3 的解释略有不同，尤其是文本文件中包含中文的时候。Python 2 的 read()方法是读取文件中指定数量的字节，对于中文可能会由于无法正常解码而出现乱码。例如，假设 sample.txt 文件的内容为“SDIBT 中国山东烟台”，那么在 Python 2.7.8 中代码运行结果如下：

```
>>> f = open( 'sample.txt', 'r')
>>> print fp.read(5)
SDIBT
>>> print fp.read(7)
Öйú□□
>>> print fp.read(8)
□ÑÌ
>>> f.close()
```

而 Python 3 对中文支持较好，对 read()方法的解释是读取文件中指定数量的字符而不是字节，中文和英文字母同等对待。对前述 sample.txt 文件，Python 3.4.2 中代码运行结果如下：

```
>>> fp = open('sample.txt','r')
>>> print(fp.read(5))
SDIBT
>>> print(fp.read(7))
中国山东烟台
>>> fp.seek(0)
0
```

```
>>>print(fp.read(8))
SDIBT 中国山
```

【例 7-3】 读取并显示文本文件所有行。

```
f =open('sample.txt', 'r')
while True:
    line=f.readline()
    if line=='':
    break
    print line,#逗号不会产生换行符,但文件中有换行符,因此会换行
f.close()
```

当然，也可以这样来写：

```
f =open(sample.txt', 'r')
li =f.readlines()
for line in li:
    print line,
f.close()
```

【例 7-4】 移动文件指针。

Python 2 和 Python 3 对于 seek()方法的理解和处理是一致的，即将文件指针定位到文件中指定字节的位置。但是由于对中文的支持程度不一样，可能会导致在 Python 2 和 Python 3 中的运行结果有所不同。例如，下面的代码在 Python 3.4.2 中运行，当遇到无法解码的字符会抛出异常。

```
>>>s ='中国山东烟台 SDIBT'
>>>fp =open(r'D:\sample.txt','w')
>>>fp.write(s)
11
>>>fp.close()
>>>fp =open(r'D:\sample.txt','r')
>>>print(fp.read(3))
中国山
>>>fp.seek(2)
2
>>>print(fp.read(1))
国
>>>fp.seek(13)
13
>>>print(fp.read(1))
D
>>>fp.seek(15)
15
>>>print(fp.read(1))
B
```

```
>>> fp.seek(3)
3
>>>print(fp.read(1))
出错信息
UnicodeDecodeError: ' gbk ' codec can ' t decode byte 0xfa in position 0: illegal
multibyte sequence
```

而在 Python 2.7.8 中,则不抛出异常,而是输出乱码,例如:

```
>>> s = '中国山东烟台 SDIBT'
>>> fp = open(r'D:\sample.txt','w')
>>> fp.write(s)
>>> fp.close()
>>> fp = open(r'D:\sample.txt','r')
>>>print(fp.read(3))
Öй
>>> fp.seek(2)
>>>print(fp.read(3))
□úÉ
>>>print(fp.read(2))
蕉
```

7.3 二进制文件操作

数据库文件、图像文件、可执行文件、音视频文件和 Office 文档等均属于二进制文件。对于二进制文件,不能使用记事本或其他文本编辑软件进行正常读写,也无法通过 Python 的文件对象直接读取和理解。必须正确理解二进制文件结构和序列化规则,才能准确地理解二进制文件内容并且设计正确的反序列化规则。序列化是指把内存中的数据在不丢失其类型信息的情况下转成对象的二进制形式表示的过程,对象序列化后的形式经过正确的反序列化过程应该能够准确地恢复为原对象。

Python 中常用的序列化模块有 struct、pickle、json、marshal 和 shelve,其中 pickle 有 C 语言实现的 cPickle,速度约提高 1000 倍,应优先考虑使用。在本节中,主要介绍 struct 和 pickle 模块在对象序列化和二进制文件操作方面的应用,其他模块请参考有关文档。

7.3.1 使用 pickle 模块

pickle 是较为常用并且速度非常快的二进制文件序列化模块,下面通过两个示例来了解如何使用 pickle 模块进行对象序列化和二进制文件读写。

【例 7-5】 使用 pickle 模块写入二进制文件。

```
import pickle
f = open('sample_pickle..dat', 'wb')
n = 7
i = 13000000
a = 99.056
```

```
s ='中国人民 123abc'
lst =[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
tu =(-5, 10, 8)
coll ={4, 5, 6}
dic ={'a':'apple', 'b':'banana', 'g':'grape', 'o':'orange'}
try:
    pickle.dump(n, f)       #表示后面将要写入的数据个数
    pickle.dump(i, f)       #把整数 i 转换为字节串,并写入文件
    pickle.dump(a, f)
    pickle.dump(s, f)
    pickle.dump(lst, f)
    pickle.dump(tu, f)
    pickle.dump(coll, f)
    pickle.dump(dic, f)
except:
    print '写文件异常!'     #如果写文件异常则跳到此处执行
finally:
    f.close()
```

【例 7-6】 读取例 7-5 中写入二进制文件的内容。

```
import pickle
f =open('sample_pickle.dat', 'rb')
n =pickle.load(f)           #读出文件的数据个数
i =0
while i<n:
    x =pickle.load(f)
    print x
    i =i+1
f.close()
```

7.3.2 使用 struct 模块

struct 也是比较常用的对象序列化和二进制文件读写模块,下面通过两个示例来简单介绍使用 struct 模块对二进制文件进行读写的用法。

【例 7-7】 使用 struct 模块写入二进制文件。

```
import struct
n =1300000000
x =96.45
b =True
s ='al@中国'
sn =struct.pack('if?', n, x, b)            #把整数 n、浮点数 x、布尔对象 b 依次转换为字节串
f =open('sample_struct.dat', 'wb')
f.write(sn)                                #写入字节串
f.write(s)                                 #字符串可直接写入
```

```
f.close()
```

【例 7-8】 使用 struct 模块读取例 7-7 写入二进制文件的内容。

```
import struct
f =open('sample_struct.dat', 'rb')
sn =f.read(9)
tu=struct.unpack('if?', sn)      #从字节串 sn 中还原出 1 个整数、1 个浮点数和 1 个布尔值,并返
                                  回元组
print(tu)
n =tu[0]
x =tu[1]
bl =tu[2]
print 'n=', n
print 'x=', x
print 'bl=', bl
s =f.read(9)
f.close()
print 's=', s
```

7.4 文件操作

Python 提供了 os、os. path 和 shutil 等大量模块支持文件级的操作。其中 os 模块的常用文件操作方法如表 7-4 所示，os. path 模块的常用文件操作方法如表 7-5 所示。

表 7-4 os 模块的常用文件操作方法

方　法	功能说明
access(path mode)	按照 mode 指定的权限访问文件
open(filename,flag[mode=0777])	按照 mode 指定的权限打开文件，默认权限为可读、可写、可执行
chmode()	改变文件的访问权限
remove(path)	删除指定的文件
rename(src,dst)	重命名文件或目录
stat(path)	返回文件的所有属性
fstat(path)	返回打开的文件的所有属性
listdir(path)	返回 path 目录下的文件和目录列表
getcwd()	返回当前工作目录

表 7-5 os. path 模块的常用文件操作方法

方　法	功能说明
abspath(path)	返回绝对路径
dirname(p)	返回目录的路径

续表

方　　法	功能说明
exists(path)	判断文件是否存在
getatime(filename)	返回文件的最后访问时间
getctime(filename)	返回文件的创建时间
getmtile(filename)	返回文件的最后修改时间
getsize(filename)	返回文件的大小
isabs(path)、isdir(path)、isfile(path)	判断 path 是否为绝对路径、目录、文件
split(path)	对路径进行分割，以列表形式返回
splitext(path)	从路径中分割文件的扩展名
splitdrive(path)	从路径中分割驱动器的名称
walk(top,func,arg)	遍历目录

下面通过几个示例来演示 os 和 os. path 模块的用法。

```
>>> import os
>>> import os.path
>>> os.path.exists('test1.txt')
False
>>> os.rename('C:\\test1.txt', 'D:\\test2.txt')   #此时'C:\\test1.txt'不存在
出错信息
>>> os.rename('C:\\dfg.txt', 'D:\\test2.txt')     #os.rename()可以实现文件的改名和移动
>>> os.path.exists('C:\\dfg.txt')
False
>>> os.path.exists('D:\\dfg.txt')
False
>>> os.path.exists('D:\\test2.txt')
True
>>> path='D:\\mypython_exp\\new_test.txt'
>>> os.path.dirname(path)
'D:\\mypython_exp'
>>> os.path.split(path)
('D:\\mypython_exp', 'new_test.txt')
>>> os.path.splitdrive(path)
('D:', '\\mypython_exp\\new_test.txt')
>>> os.path.splitext(path)
('D:\\mypython_exp\\new_test', '.txt')
```

下面的代码可以列出当前目录下所有扩展名为 pyc 的文件，其中用到了列表推导式，可以翻阅前面的 2.1.8 节了解相关知识。

```
>>> import os
>>> print [fname for fname in os.listdir(os.getcwd()) if os.path.isfile(fname) and fname.
```

```
endswith('.pyc')]
['consts.pyc', 'database_demo.pyc', 'nqueens.pyc']
```

下面的代码用来将当前目录的所有扩展名为 html 的文件修改为扩展名为 htm 的文件。

```
import os
file_list =os.listdir(".")
for filename in file_list:
    pos =filename.rindex(".")
    if filename[pos+1:] =="html":
        newname =filename[:pos+1]+"htm"
        os.rename(filename,newname)
        print(filename+"更名为："+newname)
```

shutil 模块也提供了大量的方法支持文件操作，详细的方法列表可以使用 dir(shutil)进行查看。

```
>>>import shutil
>>>dir(shutil)
['Error', 'ExecError', 'ReadError', 'RegistryError', 'SameFileError',
'SpecialFileError', '_ARCHIVE_FORMATS', '_BZ2_SUPPORTED', '_UNPACK_FORMATS',
'__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_basename', '_call_external_zip',
'_check_unpack_options', '_copyxattr', '_destinsrc', '_ensure_directory',
'_find_unpack_format', '_get_gid', '_get_uid', '_make_tarball', '_make_zipfile',
'_ntuple_diskusage', '_rmtree_safe_fd', '_rmtree_unsafe', '_samefile', '_unpack_
tarfile', '_unpack_zipfile', '_use_fd_functions', 'abspath', 'chown', 'collections',
'copy', 'copy2', 'copyfile', 'copyfileobj', 'copymode', 'copystat', 'copytree',
'disk_usage', 'errno', 'fnmatch', 'get_archive_formats', 'get_terminal_size',
'get_unpack_formats', 'getgrnam', 'getpwnam', 'ignore_patterns', 'make_archive',
'move', 'nt', 'os', 'register_archive_format', 'register_unpack_format',
'rmtree', 'stat', 'sys', 'tarfile', 'unpack_archive', 'unregister_archive_format',
'unregister_unpack_format', 'which']
```

例如，下面的代码使用该模块的 copyfile()方法复制文件。

```
>>>import shutil
>>>shutil.copyfile('C:\\dir.txt','C:\\dir1.txt')
```

7.5 目录操作

除了支持文件操作，os 和 os. path 模块还提供了大量的目录操作方法，os 模块常用的目录操作方法如表 7-6 所示，可以通过 dir(os. path)查看 os. path 模块关于目录操作的方法。

表 7-6 os 模块常用的目录操作方法

方　　法	功能说明
mkdir(path[,mode=0777])	创建目录
makedirs(path1/path2…,mode=511)	创建多级目录
rmdir(path)	删除目录
removedirs(path1/path2…)	删除多级目录
listdir(path)	返回指定目录下的文件和目录信息
getcwd()	返回当前工作目录
chdir(path)	把 path 设为当前工作目录
walk(top,topdown=True,onerror=None)	遍历目录树,该方法返回一个元组,包括 3 个元素:所有路径名、所有目录列表与文件列表

下面的代码演示了如何使用 os 模块的方法来查看、改变当前工作目录,以及创建与删除目录。

```
>>> import os
>>> os.getcwd()
'C:\\Python27'
>>> os.mkdir(os.getcwd()+'\\temp')
>>> os.chdir(os.getcwd()+'\\temp')
>>> os.getcwd()
'C:\\Python27\\temp'
>>> os.mkdir(os.getcwd()+'\\test')
>>> os.listdir('.')
['test']
>>> os.rmdir('test')
>>> os.listdir('.')
[]
```

如果需要遍历指定目录下的所有子目录和文件,可以使用递归的方法,代码如下:

```
import os
def visitDir(path):
    if not os.path.isdir(path):
        print 'Error:"',path,'" is not a directory or does not exist.'
        return
    for lists in os.listdir(path):
        sub_path = os.path.join(path, lists)
        print sub_path
        if os.path.isdir(sub_path):
            visitDir(sub_path)
visitDir('E:\\test')
```

下面的代码则使用 os 模块的 walk()方法进行指定目录的遍历。

```
import os
```

```
def visitDir2(path):
    if not os.path.isdir(path):
        print 'Error:"',path,'" is not a directory or does not exist.'
        return
    list_dirs = os.walk(path)
    for root, dirs, files in list_dirs:          #遍历该元组的目录和文件信息
        for d in dirs:
            print os.path.join(root, d)          #获取完整路径
        for f in files:
            print os.path.join(root, f)          #获取文件绝对路径
visitDir2('H:\\music')
```

也可以使用 os. path 模块的 walk()方法遍历目录,代码如下:

```
def visitDir3(arg,dirname,names):
    for filepath in names:
        print os.path.join(dirname,filepath)
os.path.walk('H:\\music',visitDir3,())
```

7.6 高级话题

在本章最后,我们一起来看几个稍微高级一点的话题,包括计算文件 MD5 值、判断文件类型和 Excel 文件读写等。

(1) 计算 CRC32 值。下面的代码分别使用 zlib 和 binascii 模块的方法来计算任意字符串的 CRC32 值,该代码经过简单修改,即可用来计算文件的 CRC32 值,相信读者可以做到这一点。

```
>>> import zlib
>>> print zlib.crc32('1234')
-1679564637
>>> print zlib.crc32('111')
1298878781
>>> print zlib.crc32('SDIBT')
2095416137
>>> import binascii
>>> binascii.crc32('SDIBT')
2095416137
```

(2) 计算文本文件中最长行的长度。

方法一:

```
f = open('D:\\test.txt','r')
allLineLens = [len(line.strip()) for line in f]
f.close()
longest = max(allLineLens)
print longest
```

方法二：

```
f = open('D:\\test.txt','r')
longest = max(len(line.strip()) for line in f)
f.close()
print longest
```

当然，为了实现这个功能，还有很多别的写法，您不妨大胆尝试一下，争取写出更加简洁、更加优雅与更加 Pythonic 的代码。

(3) 计算字符串 MD5 值。MD5 值可以用来判断文件发布之后是否被篡改，对于文件完整性保护具有重要意义。

```
>>> import hashlib
>>> import md5
>>>md5value=hashlib.md5()
>>>md5value.update('12345')
>>>md5value=md5value.hexdigest()
>>>print md5value
827ccb0eea8a706c4c34a16891f84e7b
>>>md5value=md5.md5()
>>>md5value.update('12345')
>>>md5value=md5value.hexdigest()
>>>print md5value
827ccb0eea8a706c4c34a16891f84e7b
```

上面的代码稍加改造，即可实现自己的 MD5 计算器，例如：

```
import hashlib
import os
import sys

fileName = sys.argv[1]
if os.path.isfile(fileName):
    with open(fileName,'r') as fp:
        lines = fp.readlines()
    data = ''.join(lines)
    print hashlib.md5(data).hexdigest()
```

将上面的代码保存为文件 CheckMD5OfFile.py，然后计算某文件的 MD5 值，对该文件进行微小修改后再次计算其 MD5 值，可以发现，哪怕只是修改了一点点内容，MD5 值的变换也是非常大的，如图 7-1 所示。

```
C:\Python27>echo hello world > test.txt

C:\Python27>type test.txt
hello world

C:\Python27>CheckMD5OfFile.py test.txt
edf02967b5ae298061db1f1ad2a6b6b4

C:\Python27>
C:\Python27>echo hello world1 > test.txt

C:\Python27>CheckMD5OfFile.py test.txt
04ddb543e0cf5585aa6d72420cfe2e95

C:\Python27>type test.txt
hello world1
```

图 7-1 计算文件 MD5 值

另外，也可以使用 ssdeep 工具来计算文件的模糊哈希值或分段哈希值，或者编写 Python 程序调用 ssdeep 提供的 API 函数来计算文件的模糊哈希值，

模糊哈希值可以用来比较两个文件的相似百分比。

```
>>>from ssdeep import ssdeep
>>>s =ssdeep()
>>>print s.hash_file(filename)
```

对于某些恶意软件来说,可能会对自身进行加壳或加密,真正运行时再进行脱壳或解密,这样一来,磁盘文件的哈希值和内存中脱壳或解密后进程的哈希值相差很大。因此,根据磁盘文件和其相应的进程之间模糊哈希值的相似度可以判断该文件是否包含自修改代码,并以此来判断其为恶意软件的可能性。

(4) 判断一个文件是否为 GIF 图像文件。任何一种文件都具有专门的文件头结构,在文件头中存放了大量的信息,其中就包括该文件的类型。通过文件头信息来判断文件类型的方法可以得到更加准确的信息,而不依赖于文件扩展名。

```
def is_gif(fname):
    f=open(fname,'r')
    first4=tuple(f.read(4))
    print first4
    f.close()
    return first4==('G','I','F','8')
>>>is_gif('C:\\test.gif')
('G', 'I', 'F', '8')
True
>>>is_gif('C:\\dir.txt')
False
```

(5) 比较两个文本文件是否相同。这里使用到 difflib 模块的 SequenceMatcher()方法,检测结果相对还算清晰,请大家运行下面的代码并查看结果。

```
import difflib
A =file('C:\\dir.txt','r')
B =file('C:\\dir1.txt','r')
contextA =A.read()
contextB =B.read()
s =difflib.SequenceMatcher(lambda x:x=="",contextA,contextB)
result =s.get_opcodes()
for tag,i1,i2,j1,j2 in result:
    print("%s contextA[%d:%d]=%s contextB[%d:%d]=%s"%\
        (tag,i1,i2, contextA [i1:i2],j1,j2, contextB[j1:j2]))
```

(6) 使用 xlwt 模块写入 Excel 文件。xlwt 模块默认没有安装,可以使用 pip 进行安装。

```
from xlwt import *
book =Workbook()
sheet1 =book.add_sheet("First")
al=Alignment()
```

```
al.horz=Alignment.HORZ_CENTER        #对齐方式
al.vert=Alignment.VERT_CENTER
borders=Borders()
borders.bottom=Borders.THICK         #边框样式
style=XFStyle()
style.alignment=al
style.borders=borders
row0=sheet1.row(0)
row0.write(0,'test',style=style)
book.save(r'D:\test.xls')
```

(7) 使用 xlrd 模块读取 Excel 文件,与 xlwt 模块一样,xlrd 模块也需要单独安装。

```
>>> import xlrd
>>> book = xlrd.open_workbook(r'D:\test.xls')
>>> sheet1 = book.sheet_by_name('First')
>>> row0 = sheet1.row(0)
>>> print row0[0]
text:u'test'
>>> print row0[0].value
test
```

(8) 使用 pywin32 操作 Excel 文件。pywin32 模块需要单独安装,这是一个功能非常强大的模块,提供了 Windows 底层 API 函数的封装,使得可以在 Python 中直接调用 Windows API 函数,支持大量的 Windows 底层操作,后面章节还会用到该模块的其他功能。

```
xlApp = win32com.client.Dispatch('Excel.Application')     #打开 Excel
xlBook = xlApp.Workbooks.Open('D:\\1.xls')
xlSht = xlBook.Worksheets('sheet1')
aaa = xlSht.Cells(1,2).Value
xlSht.Cells(2,3).Value = aaa
xlBook.Close(SaveChanges=1)
del xlApp
```

本章知识精要

(1) 文件操作在各类软件开发中均占有重要的地位。

(2) 文件对象的读、写方法都会自动改变文件指针的位置。

(3) Python 2. x 和 Python 3. x 对文件对象的读、写方法解释略有不同。

(4) os 和 os. path 模块提供了大量用于文件和文件夹操作的方法,包括文件和文件夹的移动、复制、删除和重命名等。

习　　题

1. 假设有一个英文文本文件,编写程序读取其内容,并将其中的大写字母变为小写字

母，小写字母变为大写字母。

2. 编写程序，将包含学生成绩的字典保存为二进制文件，然后再读取内容并显示。

3. 使用 shutil 模块中的 move()方法进行文件移动。

4. 简单解释文本文件与二进制文件的区别。

5. 编写代码，将当前工作目录修改为“C:\”，并验证，最后将当前工作目录恢复为原来的目录。

6. 编写程序，用户输入一个目录和一个文件名，搜索该目录及其子目录中是否存在该文件。

第 8 章　异常处理结构与程序调试

几乎每种高级编程语言都提供了异常处理结构。简单地说，异常是指程序运行时引发的错误，引发错误的原因有很多，如除零、下标越界、文件不存在和网络异常等。如果这些错误得不到正确的处理将会导致程序终止运行，而合理地使用异常处理结果可以使得程序更加健壮，具有更高的容错性，不会因为用户不小心的错误输入而造成程序终止。或者，也可以使用异常处理结构给用户更加友好的提示。程序出现异常后能够调试程序并快速定位和解决存在的问题是程序员综合水平和能力的重要体现。本章首先介绍 Python 异常以及异常处理结构，然后介绍几种不同的 Python 程序调试技术。

8.1　基本概念

什么是异常呢？让我们先来看个示例。

```
>>>x, y =10, 5
>>>a =x / y
>>>print A                              #拼写错误,Python 对变量名等标识符的大小写敏感
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
print A
NameError: name 'A' is not defined
>>>10 *  (1/0)                          #除 0 错误
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>>4 + spam * 3                         #使用了未定义的变量,与拼写错误的情形相似
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>>'2' +2                               #对象类型不支持特定的操作
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

在前面的章节中，您肯定已经注意到过类似的信息，没错，这就是 Python 异常的标准表现形式。熟练运用异常处理机制对于提高程序的健壮性和容错性具有重要的作用，同时也可以把 Python 晦涩难懂的错误提示转换为友好的提示显示给最终用户。

异常处理是指因为程序执行过程中出错而在正常控制流之外采取的行为。严格地说，语法错误和逻辑错误不属于异常，但有些语法错误往往会导致异常。例如，由于大小写拼写错误而试图访问不存在的对象，或者试图访问不存在的文件等。当 Python 检测到一个错

误时，解释器就会指出当前程序流已无法继续执行下去，这时候就出现了异常。当程序执行过程中出现错误时 Python 会自动引发异常，程序员也可以通过 raise 语句显式地引发异常。

需要注意的是，尽管异常处理机制非常重要也非常有效，但是不建议使用异常来代替常规的检查。例如必要的 if…else 判断等。在编程时应避免过多依赖于异常处理机制来提高程序的健壮性。

8.2 Python 异常类与自定义异常

下面的图较为全面地展示了 Python 内建异常类的继承层次。

```
BaseException
+--SystemExit
+--KeyboardInterrupt
+--GeneratorExit
+--Exception
    +--StopIteration
    +--ArithmeticError
    |    +--FloatingPointError
    |    +--OverflowError
    |    +--ZeroDivisionError
    +--AssertionError
    +--AttributeError
    +--BufferError
    +--EOFError
    +--ImportError
    +--LookupError
    |    +--IndexError
    |    +--KeyError
    +--MemoryError
    +--NameError
    |    +--UnboundLocalError
    +--OSError
    |    +--BlockingIOError
    |    +--ChildProcessError
    |    +--ConnectionError
    |    |    +--BrokenPipeError
    |    |    +--ConnectionAbortedError
    |    |    +--ConnectionRefusedError
    |    |    +--ConnectionResetError
    |    +--FileExistsError
    |    +--FileNotFoundError
    |    +--InterruptedError
    |    +--IsADirectoryError
```

```
     |    +--NotADirectoryError
     |    +--PermissionError
     |    +--ProcessLookupError
     |    +--TimeoutError
     +--ReferenceError
     +--RuntimeError
     |    +--NotImplementedError
     +--SyntaxError
     |    +--IndentationError
     |         +--TabError
     +--SystemError
     +--TypeError
     +--ValueError
     |    +--UnicodeError
     |         +--UnicodeDecodeError
     |         +--UnicodeEncodeError
     |         +--UnicodeTranslateError
     +--Warning
          +--DeprecationWarning
          +--PendingDeprecationWarning
          +--RuntimeWarning
          +--SyntaxWarning
          +--UserWarning
          +--FutureWarning
          +--ImportWarning
          +--UnicodeWarning
          +--BytesWarning
          +--ResourceWarning
```

如果需要的话，也可以继承 Python 内置异常类来实现自定义的异常类，例如：

```
class ShortInputException(Exception):
    '''您定义的异常类。'''
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast
try:
    s = raw_input('请输入 --> ')
    if len(s) < 3:
        raise ShortInputException(len(s), 3)
except EOFError:
    print '您输入了一个结束标记 EOF'
except ShortInputException, x:
    print 'ShortInputException: 输入的长度是 %d, 长度至少应是 %d' % (x.length, x.atleast)
```

```
else:
print '没有异常发生.'
```

再如下面的示例:

```
>>> class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)
>>> try:
        raise MyError(2 * 2)
    except MyError as e:
        print('My exception occurred, value:', e.value)
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

如果自己编写的某个模块需要抛出多个不同但相关的异常,可以先创建一个基类,然后创建多个派生类分别表示不同的异常。

```
class Error(Exception):                    #创建基类
    pass
class InputError(Error):                   #派生类 InputError
    """Exception raised for errors in the input.
    Attributes:
        expression -- input expression in which the error occurred message --
            explanation of the error
    """
    def __init__(self, expression, message):
        self.expression = expression
        self.message = message
class TransitionError(Error):              #派生类 TransitionError
    """Raised when an operation attempts a state transition that's not allowed.
    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """
    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message
```

8.3 Python 中的异常处理结构

异常处理结构中最常见也是最基本的结构应该是 try…except…结构。其中 try 子句中的代码块包含可能出现异常的语句，而 except 子句用来捕捉相应的异常，except 子句中的代码块则用来处理异常。如果 try 中的代码块没有出现异常则继续往下执行异常处理结构后面的代码；如果出现异常并且被 except 子句捕获则执行 except 子句中的异常处理代码；如果出现异常但没有被 except 捕获，则继续往外层抛出；如果所有层都没有捕获并处理该异常，则程序终止并将该异常抛给最终用户。该结构语法如下：

```
try:
    try 块              #被监控的语句，可能会引发异常
except Exception[, reason]:
    except 块           #处理异常的代码
```

如果需要捕获所有异常，可以使用 BaseException，即 Python 异常类的基类，代码格式如下：

```
try:
    ⋮
except BaseException, e:
    except 块           #处理所有错误
```

上面的结构可以捕获所有异常，尽管这样做很安全，但是一般并不建议这样做。对于异常处理结构，一般的建议是尽量显式捕捉可能会出现的异常并且有针对性地编写代码进行处理，因为在实际应用开发中，很难使用同一段代码去处理所有类型的异常。当然，为了避免遗漏没有得到处理的异常干扰程序的正常执行，在捕捉了所有可能想到的异常之后，也可以使用异常处理结构的最后一个 except 来捕捉 BaseException。

下面的代码演示了 try…except…结构的用法，代码运行后提示用户输入内容，如果输入的是数字则循环结束，否则一直提示用户输入正确格式的内容。

```
>>>while True:
    try:
        x = int(input("Please input a number: "))
        break
    except ValueError:
        print("That was no a valid number. Try again…")
```

在使用时，except 子句可以在异常类名字后面指定一个变量，用来捕获异常的参数或更详细的信息。

```
>>>try:
        raise Exception('spam', 'eggs')
    except Exception as inst:
        print(type(inst))       #the exception instance
        print(inst.args)        #arguments stored in .args
```

```
        print(inst)             #__str__ allows args to be printed directly,
                                #but may be overridden in exception subclasses
        x, y = inst.args        #unpack args
        print('x =', x)
        print('y =', y)
```

```
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

另外一种常用的异常处理结构是 try…except…else…语句。正如前面章节中已经提到过,这也是一种特殊形式的选择结构。如果 try 中的代码抛出了异常,并且被某个 except 捕捉则执行相应的异常处理代码,这种情况下不会执行 else 中的代码;如果 try 中的代码没有抛出异常,则执行 else 块。例如下面的代码:

```
a_list = ['China', 'America', 'England', 'France']
print '请输入字符串的序号'
while True:
    n = input()
    try:
        print a_list[n]
    except IndexError:
        print '列表元素的下标越界或格式不正确,请重新输入字符串的序号'
    else:
        break
```

在实际开发中,同一段代码可能会抛出多个异常,需要针对不同的异常类型进行相应的处理。为了支持多个异常的捕捉和处理,Python 提供了带有多个 except 的异常处理结构,这类似于多分支选择结构。一旦某个 except 捕获了异常,则后面剩余的 except 子句将不会再执行。该结构的语法如下:

```
try:
    try块                    #被监控的语句
except Exception1:
    except 块 1              #处理异常 1 的语句
except Exception2:
    except 块 2              #处理异常 2 的语句
```

下面的代码演示了该结构的用法:

```
try:
    x=input('请输入被除数: ')
    y=input('请输入除数: ')
    z=float(x) / y
except ZeroDivisionError:
```

```
    print '除数不能为零'
except TypeError:
    print '被除数和除数应为数值类型'
except NameError:
    print '变量不存在'
else:
    print x, '/', y, '=', z
```

将要捕获的异常写在一个元组中，可以使用一个 except 语句捕获多个异常，并且共用同一段异常处理代码，当然，除非确定要捕获的多个异常可以使用同一段代码来处理，否则并不建议这样做。

```
import sys
try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except (OSError, ValueError,RuntimeError, NameError):
    pass
```

最后一种常用的异常处理结构是 try…except…finally…结构。在该结构中，finally 子句中的内容无论是否发生异常都会执行，常用来做一些清理工作以释放 try 子句中申请的资源。语法如下：

```
try:
        …
    finally:
        …                       #无论如何都会执行的代码
```

例如下面的代码，无论是否发生异常，语句 print(5)都会被执行。

```
>>> try:
    3/0
except:
    print(3)
finally:
    print(5)
3
5
```

再如下面的代码，无论读取文件是否发生异常，总是能够保证正常关闭该文件。

```
try:
    f = open('test.txt', 'r')
    line = f.readline()
    print line
finally:
    f.close()
```

需要注意的一个问题是，如果 try 子句中的异常没有被捕捉和处理，或者 except 子句或 else 子句中的代码出现了异常，那么这些异常将会在 finally 子句执行完后再次抛出。例如上面的代码，使用异常处理结构的本意是为了防止文件读取操作出现异常而导致文件不能正常关闭，但是如果因为文件不存在而导致文件对象创建失败，那么 finally 子句中关闭文件对象的代码将会抛出异常从而导致程序终止运行。

```
>>>try:
    f = open('test.txt', 'r')
    line = f.readline()
    print line
finally:
    f.close()
Traceback (most recent call last):
  File "<pyshell#17>", line 6, in <module>
    f.close()
NameError: name 'f' is not defined
```

再如下面的代码，在 try 中的语句出现了异常但是没有得到处理，因此，finally 中的语句执行完以后再次抛出该异常。

```
>>>try:
    3/0
finally:
    print(5)
5
Traceback (most recent call last):
    File "<pyshell#52>", line 1, in <module>
        try:3/0
finally:
    print(5)
ZeroDivisionError: division by zero
```

下面的代码较为完整地演示了这种情况。

```
>>>def divide(x, y):
    try:
        result = x / y
    except ZeroDivisionError:
        print("division by zero!")
    else:
        print("result is", result)
    finally:
        print("executing finally clause")
>>>divide(2, 1)
result is 2.0
executing finally clause
>>>divide(2, 0)
```

```
division by zero!
executing finally clause
>>>divide("2", "1")
executing finally clause
Traceback (most recent call last):
    File "<stdin>", line 1, in ?
    File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

使用带有 finally 子句的异常处理结构时，应尽量避免在 finally 子句中使用 return 语句，否则可能会出现出乎意料的错误。例如下面的代码：

```
>>>def demo_div(a,b):
    try:
        return a/b
    except:
        pass
    finally:
        return -1
>>>demo_div(1,0)
-1
>>>demo_div(1,2)
-1
>>>demo_div(10,2)
-1
```

通过本节介绍，相信您已经了解和掌握了 Python 异常处理结构的原理和用法。简单总结一下，可以理解为“请求原谅比请求允许要容易”。也就是说，有些代码执行可能会出现错误，也可能不会出现错误，这主要由运行时的各种客观因素决定，此时建议使用异常处理结构。如果使用大量的选择结构来提前判断，仅当满足相应条件时才执行该代码，这些条件判断可能会严重干扰正常的业务逻辑，也会严重降低代码的可读性。

8.4 断言与上下文管理

断言与上下文管理可以说是两种比较特殊的异常处理方式，在形式上比异常处理结构要简单一些，能够满足简单的异常处理或条件确认，并且可以与标准的异常处理结构结合使用。

8.4.1 断言

断言语句的语法如下：

```
assert expression[, reason]
```

当判断表达式 expression 的值为真时，什么都不做；如果表达式的值为假，则抛出异常。

assert 语句一般用于对程序某个时刻必须满足的条件进行验证，仅当__debug__为 True 时有效。当 Python 脚本以-O 选项编译为字节码文件时，assert 语句将被移除以提高运行

速度。

断言和异常处理结构经常结合使用,例如:

```
>>>try:
    assert 1 ==2 , "1 is not equal 2!"
except AssertionError, reason:
    print "%s:%s"%(reason.__class__.__name__, reason)
AssertionError: 1 is not equal 2!
```

8.4.2 上下文管理

使用上下文管理语句 with 可以自动管理资源,在代码块执行完毕后自动还原进入该代码块之前的现场或上下文。不论何种原因跳出 with 块,也不论是否发生异常,总能保证资源被正确释放,这大大简化了程序员的工作,常用于文件操作、网络通信之类的场合。

with 语句的语法如下:

```
with context_expr [as var]:
    with 块
```

例如,下面的代码演示了文件操作时 with 语句的用法,使用这样的写法程序员丝毫不用担心忘记关闭文件,当文件处理完以后,将会自动关闭。

```
with open('D:\\test.txt') as f:
    for line in f:
        print line
```

8.5 用 sys 模块回溯最后的异常

当发生异常时,Python 会回溯异常,给出大量的提示,可能会给程序员的定位和纠错带来一定的困难,这时可以使用 sys 模块来回溯最后的异常。语法为

```
import sys
try:
    block
except:
    t =sys.exc_info()
    print t
```

sys.exc_info()的返回值是一个三元组(type,value/message,traceback)。其中,type 表示异常的类型,value/message 表示异常的信息或者参数,而 traceback 则包含调用栈信息的对象。

例如,下面的代码演示了其用法:

```
>>>1/0
Traceback (most recent call last):
  File "<pyshell#25>", line 1, in <module>
    1/0
```

```
ZeroDivisionError: integer division or modulo by zero>>> import sys
>>> try:
    1/0
except:
    r = sys.exc_info()
    print(r)
(<class 'ZeroDivisionError'>, ZeroDivisionError('division by zero',),
<traceback object at 0x000000000375C788>)
```

下面的代码演示了标准的异常跟踪和 sys. exc_info()之间的区别，可以看出，sys. exc_info()可以直接定位最终引发异常的原因，结果也比较简洁，但是缺点是难以直接确定引发异常的代码位置：

```
>>> def A():
    1/0
>>> def B():
    A()
>>> def C():
    B()
>>> C()
Traceback (most recent call last):
  File "<pyshell# 35>", line 1, in <module>
    C()
  File "<pyshell# 34>", line 2, in C
    B()
  File "<pyshell# 31>", line 2, in B
    A()
  File "<pyshell# 28>", line 2, in A
    1/0
ZeroDivisionError: integer division or modulo by zero
>>> try:
    C()
except:
    r = sys.exc_info()
    print r
(<type 'exceptions.ZeroDivisionError'>, ZeroDivisionError('integer division or modulo by
zero',), <traceback object at 0x0134C990>)
```

8.6 使用 IDLE 调试代码

当程序运行发生错误或者得到了非预期的结果时，是否能够熟练地对程序进行调试以快速定位和解决问题是体现程序员综合能力的重要标准之一。

几乎任何一种集成开发环境都提供了代码调试功能，Python 也不例外，Python 标准开发环境 IDLE 就提供了调试功能。使用时，首先通过单击菜单 Debug→Debugger 打开调试器窗口，然后打开并运行要调试的程序，最后切换到调试器窗口使用其中的控制按钮进行调

试。图 8-1 为 IDLE 调试窗口及其功能简要介绍。

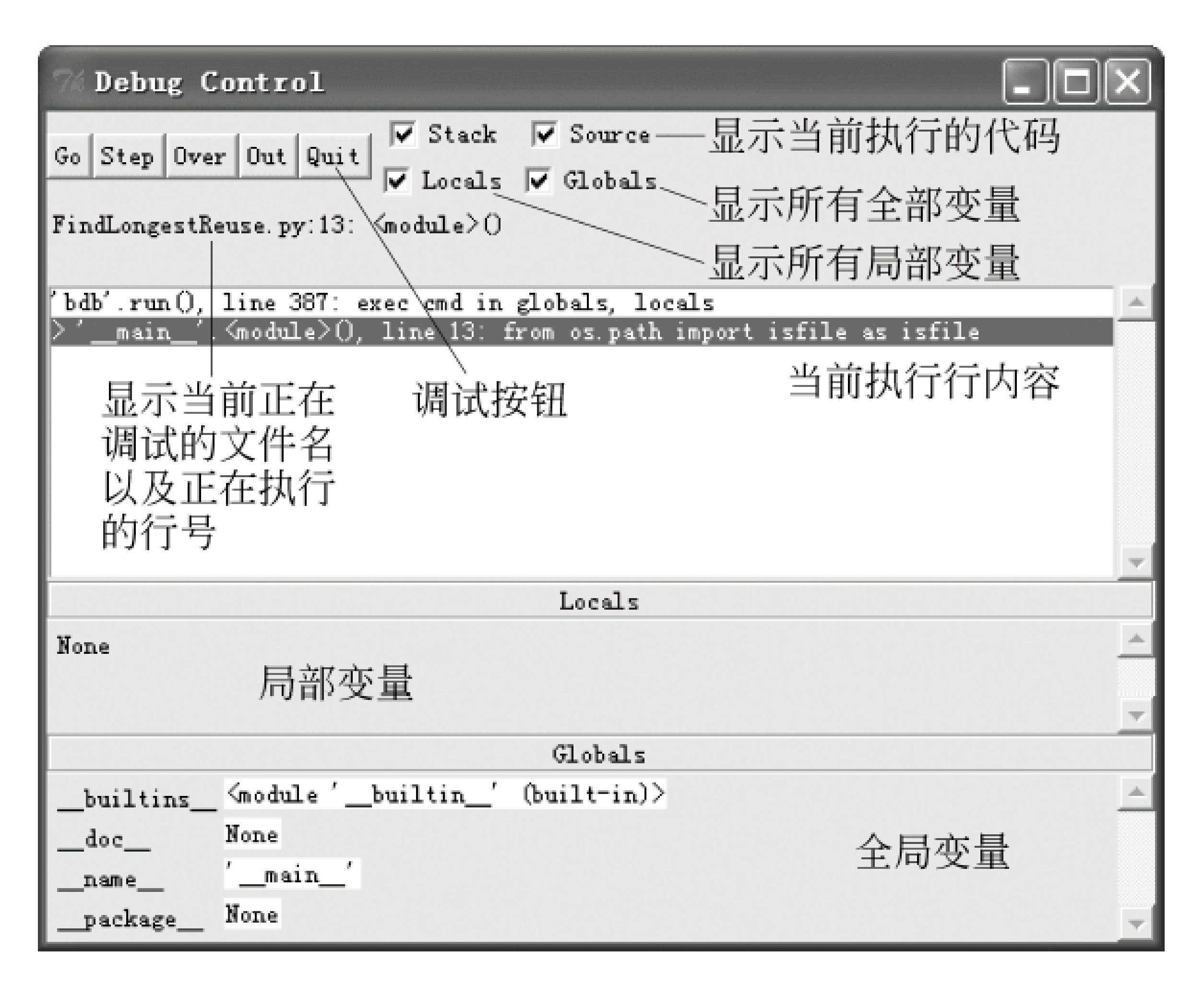


图 8-1　IDLE 调试器窗口

8.7　使用 pdb 模块调试程序

pdb 是 Python 自带的交互式源代码调试模块，源代码文件为 pdb.py，感兴趣的读者可以在 Python 安装目录下找到该文件进行阅读以理解其工作原理。pdb 模块需要导入后才能使用其中的功能，使用该模块可以完成代码调试的绝大部分功能，包括设置/清除(条件)断点、启用/禁用断点、单步执行、查看栈帧、查看变量值、查看当前执行位置、列出源代码、执行任意 Python 代码或表达式等。pdb 还支持事后调试，既可以在程序控制下被调用，也可以通过 pdb 和 cmd 接口对该调试器进行扩展。pdb 模块常用调试命令如表 8-1 所示。

表 8-1　pdb 模块常用调试命令

简写/完整命令	用法示例	解　　释
a(rgs)		显示当前函数中的参数
b(reak) [[filename:]lineno | function[, condition]]	b 173	在 173 行设置断点
	b function	在 function 函数第一条可执行语句位置设置断点
	b	不带参数则列出所有断点，包括每个断点的触发次数、当前忽略计数，以及与之关联的条件
	b 175, condition	设置条件断点，仅当 condition 的值为 True 时该断点有效
cl(ear) [filename:lineno | bpnumber [bpnumber …]]	cl	清除所有断点
	cl filename:lineno	删除指定文件指定行的所有断点
	cl 3 5 9	删除第 3、5、9 个断点

续表

简写/完整命令	用法示例	解　　释
condition bpnumber [condition]	condition 3 a<b	仅当 a<b 时 3 号断点有效
	condition 3	将 3 号断点设置为无条件断点
continue		继续运行至下一个断点或脚本结束
disable [bpnumber [bpnumber …]]	disable 3 5	禁用第 3、5 个断点，禁用后断点仍存在，可以再次被启用
d(own)		在栈跟踪器中向下移动一个栈帧
enable [bpnumber [bpnumber …]]	enable n	启用第 n 个断点
h(elp) [command]		查看 pdb 帮助
ignore bpnumber [count]		为断点设置忽略计数，count 默认值为 0。若某断点的忽略计数不为 0，则每次触发时自动减 1，当忽略计数为 0 时该断点处于活动状态
j(ump)	j 20	跳至第 20 行继续运行
l(ist) [first [,last]]	l	列出脚本清单，默认为 11 行
	l m,n	列出从第 m 行到第 n 行之间的脚本代码
	l m	列出从第 m 行开始的 11 行代码
n(ext)		执行下一条语句，遇到函数时不进入其内部
p(rint)	p i	打印变量 i 的值
q(uit)		退出 pdb 调试环境
r(eturn)		一直运行至当前函数返回
tbreak		设置临时断点，该类型断点只被中断一次，触发后该断点自动删除
step		执行下一条语句，遇到函数时进入其内部
u(p)		在栈跟踪器中向上移动一个栈帧
w(here)		查看当前栈帧
[!]statement		在 pdb 中执行语句，! 与要执行的语句之间不需要空格，任何非 pdb 命令都被解释为 Python 语句并执行，甚至可以调用函数或修改当前上下文中变量的值
		直接回车则默认执行上一个命令，但如果上一个命令是 list，则会列出接下来的 11 行代码

可以通过 3 种不同的形式来使用 pdb 模块提供的调试功能，分别为在交互模式下调试特定的代码块、在程序中显式插入断点以及把 pdb 作为模块来调试程序，接下来分别进行简要介绍。

(1) 在交互模式下使用 pdb 模块提供的功能可以直接调试语句块、表达式、函数等多种

脚本，常用的调试方法有 4 种。

① pdb. run(statement[, globals[, locals]])：调试指定语句，可选参数 globals 和 locals 用来指定代码执行的环境，默认是__main__模块的字典。

② pdb. runeval(expression[, globals[, locals]])：返回表达式的值，可选参数 globals 和 locals 的含义与上面 run()函数中的一样。

③ pdb. runcall(function[, argument, …])：调试指定函数。

④ pdb. post_mortem([traceback])：进入指定 traceback 对象的事后调试模式，如果没有指定 traceback 对象，则使用当前正在处理的一个异常。

例如，下面的代码演示了如何调试一个函数，其中“(Pdb)”为提示符，在后面输入并执行前面表 8-1 中介绍的命令即可。

```
>>> import pdb
>>> def f():
    x = 5
    print x
>>> pdb.runcall(f)
> <pyshell#5> (2)f()
(Pdb) n
> <pyshell#5> (3)f()
(Pdb) l
[EOF]
(Pdb) p x
5
(Pdb) n
5
--Return--
> <pyshell#5> (3)f()->None
(Pdb) n
>>>
```

(2) 在程序中嵌入断点来实现调试功能。

在程序中首先导入 pdb 模块，然后使用 pdb. set_trace()在需要的位置设置断点。如果程序中存在通过该方法调用显式插入的断点，那么在命令提示符环境下执行该程序或双击执行程序时将自动打开 pdb 调试环境，即使该程序当前不处于调试状态。例如下面的程序：

```
IsPrime.py:
import pdb
n=37
pdb.set_trace()
for i in range(2,n):
    if n%i==0:
        print 'No'
        break
```

```
else:
    print 'Yes'
```

使用 pdb 设置的断点，运行后自动打开调试模式，如图 8-2 所示。

```
>>> ================================ RESTART =
>>>
> C:\python27\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) n
> C:\python27\isprime.py(6)<module>()
-> if n%i==0:
(Pdb) n
> C:\python27\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) p i
2
(Pdb) n
> C:\python27\isprime.py(6)<module>()
-> if n%i==0:
(Pdb) n
> C:\python27\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) p i
3
(Pdb)
```

图 8-2 自动打开调试模式

在命令提示符环境中运行该程序同样自动打开调试模式，如图 8-3 所示。

```
C:\WINDOWS\system32\cmd.exe - IsPrime.py
C:\Documents and Settings\Administrator>cd \python27

C:\Python27>IsPrime.py
> c:\python27\isprime.py(5)<module>()
-> for i in range(2,n):
(Pdb) p i
*** NameError: NameError("name 'i' is not defined",)
(Pdb) n
> c:\python27\isprime.py(6)<module>()
-> if n%i==0:
(Pdb) p i
2
(Pdb) l
  1     import pdb
  2
  3     n=37
  4     pdb.set_trace()
  5     for i in range(2,n):
  6  ->     if n%i==0:
  7             print 'No'
  8             break
  9     else:
 10         print 'Yes'
[EOF]
(Pdb)
```

图 8-3 自动打开调试模式

(3) 使用命令行调试程序。

在命令行提示符下执行“python -m pdb 脚本文件名”，则直接进入调试环境；当调试结束或程序正常结束以后，pdb 将重启该程序。例如，把上面的程序 IsPrime. py 中 pdb 模块的导入和断点插入函数都删除，然后在命令提示符环境中使用调试模式运行，如图 8-4 所示。

```
C:\Python27>IsPrime.py
Yes

C:\Python27>python -m pdb IsPrime.py
> C:\python27\isprime.py(1)<module>()
-> n=37
(Pdb) n
> C:\python27\isprime.py(2)<module>()
-> for i in range(2,n):
(Pdb) w
  C:\python27\lib\bdb.py(400)run()
-> exec cmd in globals, locals
  <string>(1)<module>()
> C:\python27\isprime.py(2)<module>()
-> for i in range(2,n):
(Pdb) n
> C:\python27\isprime.py(3)<module>()
-> if n%i==0:
(Pdb) p i
2
(Pdb) p n
37
(Pdb)
```

图 8-4　以调试模式运行程序

本章知识精要

（1）异常处理结构可以提高程序的容错性和健壮性，但不建议过多依赖异常处理结构。

（2）可以继承 Python 内建异常类来实现自定义的异常类。

（3）可以使用 BaseException 来捕获所有异常，但不建议这样做。

（4）异常处理结果中也可以使用 else 子句，当没有异常发生时执行 else 子句中的代码块。

（5）断言语句 assert 一般用于对程序某个时刻必须满足的条件进行验证。

（6）上下文管理语句 with 在代码块执行完毕后能够自动还原进入代码块之前的现场或上下文，不论是否发生异常总能保证资源被正确释放。

（7）异常处理结构的 finally 子句中的代码仍可能会抛出异常。

（8）可以通过 3 种方式使用 pdb 模块的调试功能：在交互模式下使用 pdb 模块的方法调试指定函数或语句、在程序中显式插入断点、执行 Python 程序时指定 pdb 调试模块。

习　　题

1. Python 异常处理结构有哪几种形式？
2. 异常和错误有什么区别？
3. 使用 pdb 模块进行 Python 程序调试主要有哪几种用法？
4. Python 内建异常类的基类是____________________。
5. 断言语句的语法为____________________。
6. Python 的上下文管理语句为____________________。

第二篇　Python 高级编程与应用

在本书第一篇中，重点介绍了 Python 的基础知识，包括各种基本数据结构、控制结构、类与函数的编写、文件操作以及异常处理结构和 Python 程序调试技术等。第一篇是最重要的一部分知识，也是学习和熟练运用大量扩展库的必备知识。虽然说 Python 就像胶水一样能够把很多不同语言黏合在一起，虽然说 Python 更多的工作需要借助于扩展库来实现，但是 Python 基础知识的掌握和理解对于任何应用开发工作都具有非常重要的作用。

在理解和掌握并能够熟练运行 Python 基础知识以后，本书第二篇则把重点放在了一些扩展库的应用上。当然，Python 扩展库几乎涉及所有领域和行业，一本书也不可能给出非常全面的介绍，作者仅选取了部分领域和扩展库进行了入门级的介绍，更多的内容还需要读者查阅帮助文档以满足实际开发中的需求。同时，在同一个领域也存在不同的扩展库，这些扩展库各有特点，可能需要读者根据具体的需要以及所使用的操作系统平台和 Python 版本进行甄别和选择。

在第 1 章曾经提到过，pip 是管理 Python 扩展库的重要工具，绝大多数的扩展库都可以使用 pip 来安装、升级和卸载，读者需要做的仅仅是在保证联网的情况下执行几条命令而已，这一点对本书后面的绝大部分扩展库都是适用的。如果您遇到无法通过 pip 安装或者安装之后无法使用的扩展库，很可能该扩展库需要使用独立的安装包来安装，或者还依赖一些 dll 文件，此时建议您登录其官方网站下载安装包或依赖文件。

第 9 章　GUI 编 程

在前面所有章节以及后面的大部分章节中，我们都是使用控制台应用程序的编写来演示 Python 语言的精妙和强大。然而，最终用户可能更习惯使用 GUI 程序，毕竟现在已经很少有终端用户使用控制台的命令行工作模式了，他们更希望看到带有窗口、按钮、菜单、组合框、列表框等 GUI 元素的应用程序，也许这也正是读者的需求。为此，本章介绍了 Python 的 GUI 编程，让读者的程序更容易使用和操作。

目前，常用 GUI 工具除了 Python 标准 GUI 库 Tkinter，还有功能强大的跨平台库 wxPython、基于 Java 的 Jython、支持.NET 应用程序的 IronPython 等。本章以 wxPython 为例来介绍 Python 的 GUI 应用开发，有了 wxPython 的基础，相信读者也能很快掌握和使用其他 GUI 库。

在本书编写时，wxPython 的最新版本是 3.0，读者可以登录网页 http://wxpython.org/download.php 进行下载并安装。

使用 wxPython 创建 GUI 程序的 3 个主要步骤如下。

（1）导入 wxPython 包。

（2）建立框架类。框架类的父类为 wx.Frame，在框架类的构造函数中调用父类的构造函数进行初始化；然后为 frame 类添加各种控件以及事件处理方法，如需在窗体上增加其他控件，可在构造函数中增加代码，如需处理相应事件，可增加框架类的成员函数，并将事件处理方法与相应的控件进行绑定。

（3）建立主程序。通常需要做 4 件事——创建应用程序对象、创建框架类对象、显示框架、开始事件循环。执行 frame.Show(True)后，框架才能看得见，执行 app.MainLoop( )后，框架才能处理事件。

9.1　Frame

Frame 也称为框架或窗体，它是所有框架的父类，也是包含标题栏、菜单、按钮等其他控件的容器，运行之后可移动、缩放。

创建 GUI 程序框架时，需要使用继承 wx.Frame 派生出子类，在派生类中调用基类构造函数进行必要的初始化，其构造函数格式如下：

```
__init__(self, Window parent, int id=-1, String title=EmptyString, Point pos=
DefaultPosition, Size size=DefaultSize, long style=DEFAULT_FRAME_STYLE, String name=
FrameNameStr)
```

各参数具体含义如下。

（1）parent：框架的父窗体。该值为 None 时表示创建顶级窗体。

（2）id：新窗体的 wxPython ID 号。可以明确地传递一个唯一的 ID，也可传递－1，这

时 wxPython 将自动生成一个新的 ID，由系统来保证其唯一性。

(3) title：窗体的标题。

(4) pos：wx. Point 对象，用来指定新窗体的左上角在屏幕中的位置。通常(0,0)是显示器的左上角坐标。当将其设定为 wx. DefaultPosition 的，其值为(−1,−1)，表示让系统决定窗体的位置。

(5) size：wx. Size 对象，用来指定新窗体的初始大小。当将其设定为 wx. DefaultSize 时，其值为(−1,−1)，表示由系统来决定窗体的初始大小。

(6) style：指定窗体的类型的常量，wx. Frame 的常用样式如表 9-1 所示。对一个窗体控件可以同时使用多个样式，使用“位或”运算符 | 连接即可。比如 wx. DEFAULT_FRAME_STYLE 样式就是由以下几个基本样式的组合：wx. MAXIMIZE_BOX | wx. MINIMIZE_BOX | wx. RESIZE_BORDER | wx. SYSTEM_MENU | wx. CAPTION | wx. CLOSE_BOX。要从一个组合样式中去掉个别的样式可以使用^按位异或操作符，例如，要创建一个默认样式的窗体，但要求用户不能缩放和改变窗体的尺寸，可以使用这样的组合：wx. DEFAULT_FRAME_STYLE^(wx. RESIZE_BORDER | wx. MAXIMIZE_BOX | wx. MINIMIZE_BOX)。

表 9-1　wx. Frame 的常用样式

样　式	说　明
wx. CAPTION	增加标题栏
wx. DEFAULT_FRAME_STYLE	默认样式
wx. CLOSE_BOX	标题栏上显示“关闭”按钮
wx. MAXIMIZE_BOX	标题栏上显示“最大化”按钮
wx. MINIMIZE_BOX	标题栏上显示“最小化”按钮
wx. RESIZE_BORDER	边框可改变尺寸
wx. SIMPLE_BORDER	边框没有装饰
wx. SYSTEM_MENU	增加系统菜单(有“关闭”、“移动”、“改变尺寸”等功能)
wx. FRAME_SHAPED	用该样式创建的框架可以使用 SetShape()方法来创建一个非矩形的窗体
wx. FRAME_TOOL_WINDOW	给框架一个比正常小的标题栏，使框架看起来像一个工具框窗体

(7) name：框架的名字，指定后可以使用这个名字来寻找这个窗体。

可以看到，wx. Frame. __init__( )方法只有一个参数 parent 没有默认值，因而最简单的调用方式是

```
wx.Frame.__init__(self, parent=None)
```

这将生成一个默认位置、默认大小、默认标题的顶层窗体。

在初始化窗体时可以明确给构造函数传递一个正整数作为新窗体的 ID，此时由程序员自己来保证 ID 不重复并且没有与预定义的 ID 号冲突。例如，不能使用 wx. ID_OK (5100)、wx. ID_CANCEL(5101)、wx. ID_ANY(−1)、wx. ID_COPY(5032)和 wx. ID_

APPLY(5102)等预定义 ID 号对应的数值。

如果您无法确定使用哪个数值作为 ID 的话,可以使用 wx.NewId()函数来生成 ID 号,这样就可以避免确保 ID 号唯一性的麻烦。

```
id = wx.NewId()
frame = wx.Frame.__init__(None, id)
```

当然,也使用全局常量 wx.ID_ANY(值为－1)来让 wxPython 自动生成新的唯一 ID 号,需要时可以使用 GetId()方法来得到它,例如:

```
frame = wx.Frame.__init__(None, -1)
id = frame.GetId()
```

在本节的最后,我们通过一个具体的示例来演示使用 wxPython 创建 GUI 应用程序的思路,将下面的代码保存并运行,则会在窗体上的文本框中动态显示鼠标相对于窗体(即窗体左上角坐标为(0,0))的当前位置,可以移动鼠标并观察值的变化。

```
import wx
class Frame0(wx.Frame):
    def __init__(self,superior):
        wx.Frame.__init__(self,parent=superior, title=u'My First Form', size=(300, 300))
        panel = wx.Panel(self)
        panel.Bind(wx.EVT_MOTION, self.OnMove)          #绑定事件处理函数
        wx.StaticText(parent=panel,label="Pos:", pos=(10, 20))
        self.posCtrl = wx.TextCtrl(parent=panel,pos=(40, 20))
    def OnMove(self, event):                              #鼠标移动事件处理函数
        pos = event.GetPosition()
        self.posCtrl.SetValue("%s, %s"
        % (pos.x, pos.y))
if __name__ == '__main__':
    app = wx.App()
    frame = Frame0(None)
    frame.Show(True)
    app.MainLoop()
```

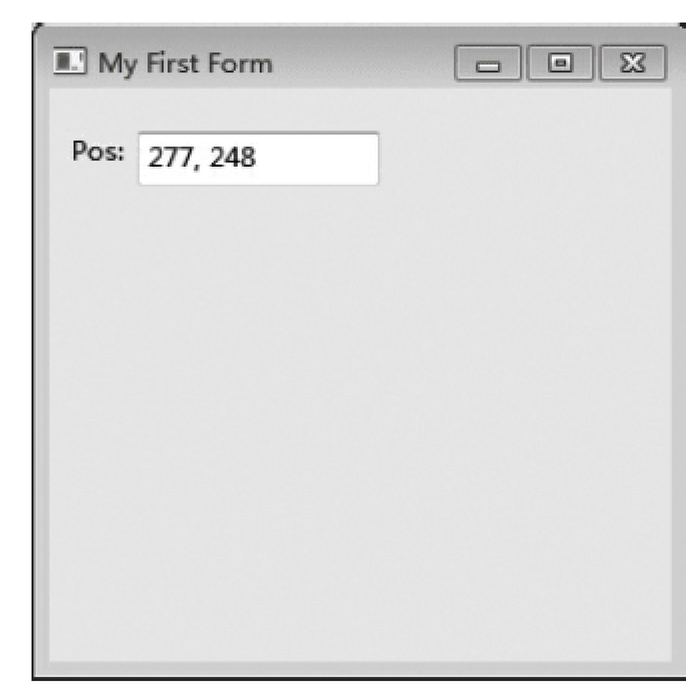


图 9-1　显示鼠标位置

运行结果如图 9-1 所示,当鼠标在窗体内移动时,文本框内实时显示鼠标当前坐标,当鼠标移动到窗体之外时,文本框中的数值将不再变化。

9.2　Controls

wxPython 提供了几乎所有常用的控件,如按钮、静态文本标签、文本框、单选按钮、复选框、对话框、菜单、列表框和树形控件等。如需在窗体上增加其他控件,可在窗体构造函数中增加代码;如需响应和处理特定事件,可增加框架类的成员函数,并进行相应的绑定操作。在本节中,通过几个示例来演示 wxPython 控件的用法。为了方便介绍,大致根据控件的类

型将控件进行了不同的组合放在一起介绍，而不是孤零零地单个介绍控件的用法。

9.2.1 Button、StaticText 和 TextCtrl

按钮控件的构造函数语法如下：

```
__init__(self, Window parent, int id=-1, String label=EmptyString, Point pos=DefaultPosition, Size size=DefaultSize, long style=0, Validator validator=DefaultValidator, String name=ButtonNameStr)
```

按钮主要用来接受用户的单击操作，而按钮上面的文本一般是创建时直接指定的，很少需要修改。当然，如果确实需要动态修改的话，可以通过 SetLabelText()方法来实现这个目的，再结合 GetLabelText()方法来获取按钮控件上面显示的文本，则可以实现同一个按钮完成不同功能的目的。为按钮绑定事件处理函数的方法为

```
Bind(event, handler, source=None, id=-1, id2=-1)
```

静态文本控件主要用来显示文本或给用户操作提示，不接受用户单击或双击事件，可以使用 SetLabel()方法动态为 StaticText 控件设置文本。静态文本控件的构造函数语法如下：

```
__init__(self, Window parent, int id=-1, String label=EmptyString, Point pos=DefaultPosition, Size size=DefaultSize, long style=0, String name=StaticTextNameStr)
```

文本框主要用来接受用户的文本输入，可以使用 GetValue()方法获取文本框中输入的内容，使用 SetValue()方法设置文本框中的文本，文本框控件的构造函数语法如下：

```
__init__(self, Window parent, int id=-1, String value=EmptyString, Point pos=DefaultPosition, Size size=DefaultSize, long style=0, Validator validator=DefaultValidator, String name=TextCtrlNameStr)
```

下面通过一个示例来演示这 3 个控件的用法，将下面的代码保存为 wxIsPrime.py，运行后用户输入一个整数，单击按钮后判断是否为素数并输出结果。

```
import wx
from math import sqrt

class IsPrimeFrame(wx.Frame):
    def __init__(self, superion):
        wx.Frame.__init__(self, parent=superion, title='Check Prime', size=(400,200))
        panel =wx.Panel(self)
        panel.SetBackgroundColour('Yellow')        #设置窗体背景颜色
        wx.StaticText(parent=panel, label='Input a integer:', pos=(10,10))
                                                               #添加静态文本控件
        self.inputN =wx.TextCtrl(parent=panel, pos=(120,10))          #添加文本框
        self.result =wx.StaticText(parent=panel, label='', pos=(10,50))
        self.buttonCheck =wx.Button(parent=panel, label='Check', pos=(70,90))
                                                               #添加按钮
```

```
        #为按钮绑定事件处理方法
        self.Bind(wx.EVT_BUTTON, self.OnButtonCheck, self.buttonCheck)
        self.buttonQuit =wx.Button(parent=panel, label='Quit', pos=(150,90))
        self.Bind(wx.EVT_BUTTON, self.OnButtonQuit, self.buttonQuit)

    def OnButtonCheck(self, event):
        self.result.SetLabel('')
        try:
            num =int(self.inputN.GetValue())         #获取用户输入的数字
        except BaseException,e:
            self.result.SetLabel('not a integer')
            return
        n =int(sqrt(num))
        for i in range(2,n+1):                       #判断用户输入的数字是否为素数
            if num%i ==0:
                self.result.SetLabel('No')          #使用静态文本框显示结果
                break
        else:
            self.result.SetLabel('Yes')

    def OnButtonQuit(self, event):
        dlg=wx.MessageDialog(self,'Really Quit?','Caution',\
            wx.CANCEL|wx.OK|wx.ICON_QUESTION)
        if dlg.ShowModal() ==wx.ID_OK:
            self.Destroy()
if __name__ =='__main__':
    app =wx.App()
    frame =IsPrimeFrame(None)
    frame.Show()
    app.MainLoop()
```

运行结果如图 9-2 所示。

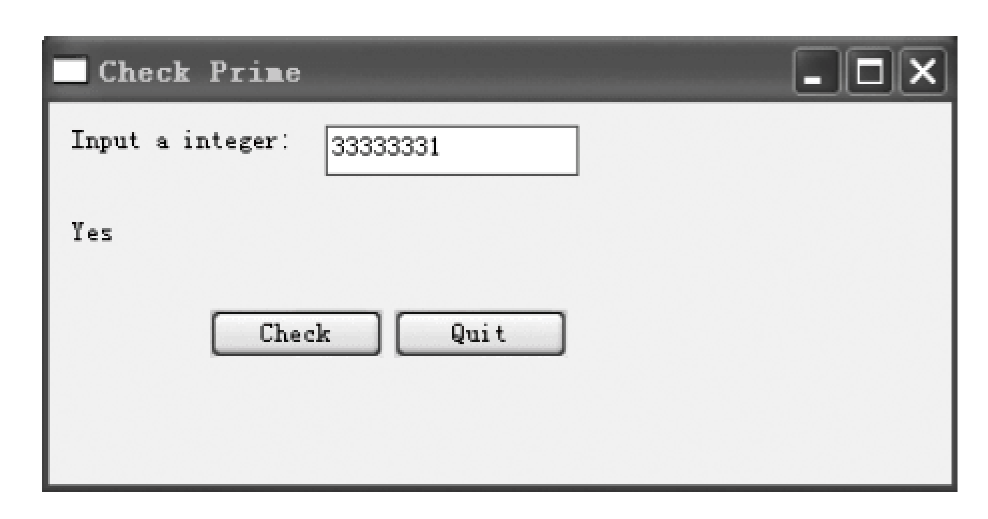


图 9-2　素数判断

9.2.2　Menu

菜单可以分为普通菜单和弹出式菜单两大类，其中普通菜单也就是大多数窗口菜单栏的下拉菜单，弹出式菜单也称为上下文菜单，一般需要使用鼠标右键激活，并根据不同的环

境或上下文来显示不同的菜单项。下面对这两类菜单分别进行介绍。

1. 创建普通菜单

```
self.frame =wx.Frame(parent=None, title='wxGUI', size=(640,480))
self.panel =wx.Panel(self.frame, -1)
self.menuBar =wx.MenuBar()                           #创建菜单栏
self.menu =wx.Menu()                                 #创建菜单
self.menuOpen =self.menu.Append101, 'Open')          #创建菜单项
self.menuSave =self.menu.Append(102, 'Save')
self.menuSaveAs =self.menu.Append(103, 'Save As')
self.menu.AppendSeparator()                          #分隔符
self.menuClose =self.menu.Append(104, 'Close')
self.menuBar.Append(self.menu, '&File')              #将菜单添加至菜单栏
self.menu =wx.Menu()
self.menuCopy =self.menu.Append(201, 'Copy')
self.menuCut =self.menu.Append(202, 'Cut')
self.menuPaste =self.menu.Append(203, 'Paste')
self.menuBar.Append(self.menu, '&Edit')
```

创建菜单完成之后,通过下面的代码将创建的菜单设置为窗体菜单。

```
self.frame.SetMenuBar(self.menuBar)
```

2. 创建弹出式菜单

```
self.popupMenu =wx.Menu()                            #创建菜单
self.popupCopy =self.popupMenu.Append(901, 'Copy')   #创建菜单项
self.popupCut =self.popupMenu.Append(902, 'Cut')
self.popupPaste =self.popupMenu.Append(903, 'Paste')
```

接下来为窗体绑定右击操作,

```
self.Bind(wx.EVT_RIGHT_DOWN, self.OnRClick)
```

然后编写右击处理函数,用户右击时弹出上面定义的弹出式菜单。

```
def OnRClick(self, event):
    pos = (event.GetX(),event.GetY())                #获取鼠标当前位置
    self.panel.PopupMenu(self.popupMenu, pos)            #在鼠标当前位置弹出上下文菜单
```

3. 为菜单项绑定单击事件处理函数

对于普通下拉式菜单和弹出式菜单,为菜单项绑定事件处理函数的方式是一样的,例如下面的代码,其中第二个数值型的参数是菜单项的 ID,最后一个参数是事件处理函数的名称。绑定之后,运行程序并单击某菜单项,则会执行相应的事件处理函数中的代码。

```
wx.EVT_MENU(self, 102, self.OnOpen)
wx.EVT_MENU(self, 103, self.OnSave)
wx.EVT_MENU(self, 104, self.OnSaveAs)
wx.EVT_MENU(self, 105, self.OnClose)
```

4. 编写菜单项的单击事件处理函数

具体的事件处理函数根据不同的业务逻辑有所不同，这里仅演示如何在状态栏上显示一段文本，有关状态栏的介绍请参考 9.2.3 节。

```
def OnNew(self, event):
    self.statusBar.SetStatusText('You clicked the New menu.')
```

9.2.3 ToolBar 和 StatusBar

工具栏往往用来显示当前上下文最常用的功能按钮，一般而言，工具栏按钮是菜单全部功能的子集。状态栏主要用来显示当前状态或给用户友好提示，例如，Word 软件中的状态栏上显示的当前页码、总页数、节数以及当前行与当前列等信息。下面我们分别介绍这两个控件的创建和使用方法。

1. 创建工具栏

```
self.toolbar = self.frame.CreateToolBar()
```

接下来在工具栏上添加工具，相应的工具栏图片需要提前准备好，并存放于当前目录下。

```
self.toolbar.AddSimpleTool(9999,wx.Image('open.png',wx.BITMAP_TYPE_PNG).ConvertToBitmap
(),'Open', 'Click to Open a file')
```

然后使用下面的代码使工具栏有效。

```
self.toolbar.Realize()
```

最后绑定事件处理函数，事件处理函数的编写与前面介绍的按钮、菜单项等控件的事件处理函数一样，不再赘述。

```
wx.EVT_TOOL(self, 9999, self.OnOpen)
```

2. 创建状态栏

状态栏的创建和使用相对比较简单，通过下面的代码即可创建：

```
self.statusBar = self.frame.CreateStatusBar()
```

如果需要在状态栏上显示状态或者显示文本以提示用户，可以通过下面的代码设置状态栏文本：

```
self.statusBar.SetStatusText('You clicked the Open menu.')
```

9.2.4 对话框

wxPython 提供了一整套丰富的预定义对话框支持友好界面开发，常用的对话框有以下 7 种。

(1) MessageBox：简单消息框。

(2) GetTextFromUser：接受用户输入的文本。

(3) GetPasswordFromUser：接受用户输入的密码。

(4) GetNumberFromUser：接受用户输入的数字。

(5) FileDialog:“文件”对话框。

(6) FontDialog:“字体”对话框。

(7) ColourDialog:“颜色”对话框。

除用于信息提示的简单消息框之外,其他几种对话框的使用都遵循固定的步骤:首先创建对话框,其次显示对话框,最后根据对话框的返回值采取不同的操作。下面的代码演示了 MessageBox 的用法,完整代码可以参照 9.2.5 节的示例。

```
wx.MessageBox(finalStr)
```

下面的代码演示了 MessageDialog 的用法,完整代码请参考 9.2.1 小节。其他对话框可以参考 MessageDialog 对话框的用法。

```
def OnButtonQuit(self, event):
    dlg=wx.MessageDialog(self,'Really Quit?', 'Caution', wx.CANCEL|wx.OK| \
                         wx.ICON_QUESTION)
    if dlg.ShowModal()==wx.ID_OK:
        self.Destroy()
```

9.2.5 RadioButton、CheckBox 和 ComboBox

单选按钮常用来实现用户在多个选项中的互斥选择,在同一组内多个选项中只能选择一个,当选择发生变化之后,之前选择的选项自动失效。单选钮控件的构造函数语法如下:

```
__init__(self, Window parent, int id=-1, String label=EmptyString, Point pos=
DefaultPosition, Size size = DefaultSize, long style = 0, Validator validator =
DefaultValidator, String name=RadioButtonNameStr)
```

可以使用 wxPython 的 SashWindow 控件对单选钮进行分组,也可以使用单选钮控件的样式进行分组,每组的第一个单选钮使用 wx. RB_GROUP 样式,其他单选框不使用该样式。

复选框往往用来实现非互斥多选的功能,多个复选框之间的选择互不影响。复选框的构造函数语法如下:

```
__init__(self, Window parent, int id=-1, String label=EmptyString, Point pos=
DefaultPosition, Size size=DefaultSize, long style=0, Validator
validator=DefaultValidator, String name=CheckBoxNameStr)
```

单选按钮和复选框的很多操作是通用的,例如,可以使用 GetValue()方法判断单选按钮是否被选中,使用 SetValue(True)将单选按钮设置为选中状态,使用 SetValue(False)将单选按钮设置为未选中状态。

组合框用来实现从固定的多个选项中选择其中一个的操作,外观与文本框类似,但是单击下拉箭头时弹出所有可选项,极大地方便了用户的操作,并且在窗体上不占用太大的空间。组合框的构造函数语法如下:

```
__init__(Window parent, int id=-1, String value=EmptyString, Point pos=DefaultPosition,
Size size = DefaultSize, List choices = EmptyList, long style = 0, Validator validator =
DefaultValidator, String name=ComboBoxNameStr)
```

在某些应用中,可能需要响应单选按钮、复选框以及组合框的单击事件,根据不同的需要可以使用 wx.EVT_RADIOBOX()、wx.EVT_CHECKBOX()和 wx.EVT_COMBOBOX()分别为单选按钮、复选框和组合框来绑定事件处理函数。

下面的代码演示了这 3 个控件的用法。

```
import wx
class wxGUI(wx.App):
    def OnInit(self):
        self.frame =wx.Frame(parent=None, title='wxGUI', size=(640,480))
        self.panel =wx.Panel(self.frame, -1)

        self.radioButtonSexM =wx.RadioButton(self.panel, -1, 'Male', pos=(280,160))
        self.radioButtonSexF =wx.RadioButton(self.panel, -1, 'Female', pos=(280,180))
        self.checkBoxAdmin =wx.CheckBox(self.panel, -1, 'Aministrator', pos=(350,180))

        self.label1 =wx.StaticText(self.panel, -1, 'UserName:', pos=(200,210), style=wx.
        ALIGN_RIGHT)
        self.label2 =wx.StaticText(self.panel, -1, 'Password:', pos=(200,230), style=wx.
        ALIGN_RIGHT)

        self.textName =wx.TextCtrl(self.panel, -1, pos=(270,210), size=(160,20))
        self.textPwd =wx.TextCtrl(self.panel, -1, pos=(270,230), size=(160,20), style=wx.
        TE_PASSWORD)      #密码文本框

        self.buttonOK =wx.Button(self.panel, -1, 'OK', pos=(230,260))
        self.Bind(wx.EVT_BUTTON, self.OnButtonOK, self.buttonOK)
        self.buttonCancel =wx.Button(self.panel, -1, 'Cancel', pos=(320,260))
        self.Bind(wx.EVT_BUTTON, self.OnButtonCancel, self.buttonCancel)
        self.buttonOK.SetDefault()

        #ComboBox
        self.comboBox =wx.ComboBox(self.panel, value='Click here',\
                                   choices=['a','b','c'],\
                                   pos=(200,100), size=(100,30))
        self.Bind(wx.EVT_COMBOBOX, self.OnCombo, self.comboBox)

        self.frame.Show()
        return True

    def OnCombo(self, event):   #组合框单击事件处理函数
        wx.MessageBox(self.comboBox.GetValue())

    def OnButtonOK(self, event):   #根据用户输入弹出提示
        finalStr =''
        if self.radioButtonSexM.GetValue() ==True:
            finalStr +='Sex:Male\n'
        elif self.radioButtonSexF.GetValue() ==True:
```

```
            finalStr += 'Sex:Female\n'
        if self.checkBoxAdmin.GetValue() == True:
            finalStr += 'Administrator\n'
        if self.textName.GetValue()== 'dongfuguo' and self.textPwd.GetValue()
        == 'dongfuguo':
            finalStr += 'user name and password are correct\n'
        else:
            finalStr += 'user name or password is incorrect\n'
        wx.MessageBox(finalStr)
    def OnButtonCancel(self, event):    #清空用户输入
        self.radioButtonSexM.SetValue(True)
        self.radioButtonSexF.SetValue(False)
        self.checkBoxAdmin.SetValue(True)
        self.textName.SetValue('')
        self.textPwd.SetValue('')

app = wxGUI()
app.MainLoop()
```

上面的代码运行结果如图 9-3 所示，单击组合框并改变选择之后会自动弹出消息框提示选择的项，选择单选按钮、复选框并输入文本框中要求的用户名和密码之后，单击 OK 按钮会弹出消息框提示输入和选择的内容，单击 Cancel 按钮自动清除用户的输入，并默认将单选按钮 Male 设置为选中状态。

9.2.6 ListBox

列表框用来放置多个元素并提供给用户进行选择，其中每个元素都是字符串，支持用户单选和多选。列表框的常用样式和常用方法如表 9-2 和表 9-3 所示。

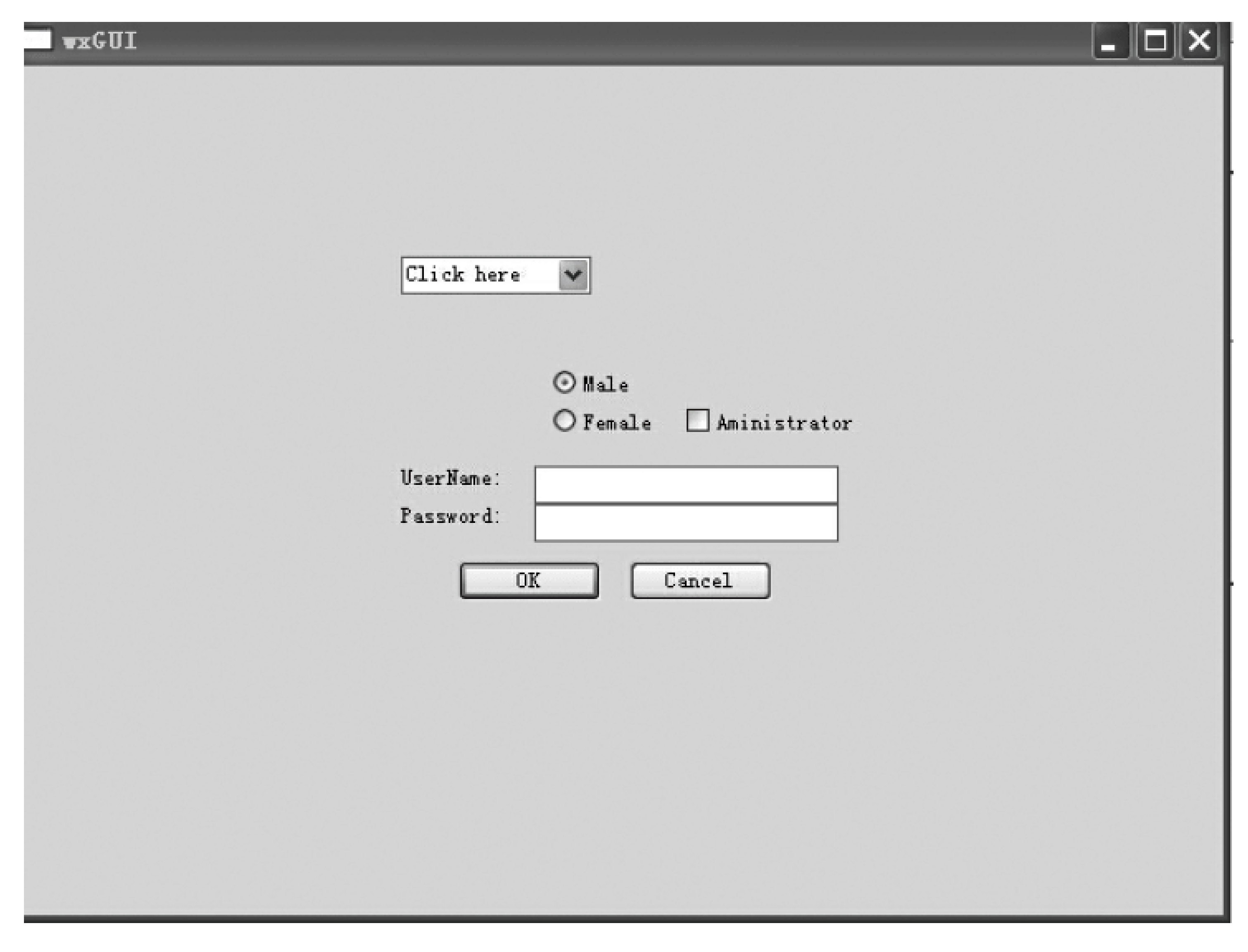


图 9-3 单选按钮、复选框和组合框的用法

表 9-2　列表框的常用样式

样　　式	说　　明
wx.LB_EXTENDED	可以使用 Shift 键和鼠标配合选择连续多个元素
wx.LB_MULTIPLE	可以选择多个不连续的元素
wx.LB_SINGLE	最多只能选择一个元素
wx.LB_ALWAYS_SB	始终显示一个垂直滚动条
wx.LB_HSCROLL	仅在需要时显示一个垂直滚动条
wx.LB_SORT	列表框中的元素按字母顺序排序

表 9-3　列表框的常用方法

方　法　名	说　　明
Append(string)	在列表框尾部增加一个元素
Clear()	删除列表框中的所有元素
Delete(index)	删除列表框指定索引的元素
FindString(string)	返回指定元素的索引，没找到则返回－1
GetCount()	返回列表框中元素的个数
GetSelection()	返回当前选择项的索引，仅对单选列表框有效
SetSelection(index, True/False)	设置指定索引的元素的选择状态
GetStringSelection()	返回当前选择的元素，仅对单选列表框有效
GetString(index)	返回指定索引的元素
SetString(index, string)	设置指定索引的元素文本
GetSelections()	返回包含所选元素的元组
InsertItems(items,pos)	在指定位置之前插入元素
IsSelected(index)	返回指定索引的元素的选择状态
Set(choices)	使用列表 choices 的内容重新设置列表框

下面的代码演示了列表框的用法，运行程序后，列表框中显示周日到周六的每天，用户单击其中一个后弹出一个消息框来提示所选择的内容，单击 Quit 按钮时弹出关闭前的确认对话框。

```
import wx

class ListBoxDemo(wx.Frame):
    def __init__(self, superion):
        wx.Frame.__init__(self, parent=superion, title='ListBox demo', size=(200,200))
        panel =wx.Panel(self)
        self.buttonQuit =wx.Button(parent=panel, label='Quit', pos=(60,120))
        self.Bind(wx.EVT_BUTTON, self.OnButtonQuit, self.buttonQuit)
```

```
        li = ['Sunday','Monday','Tuesday','Wednesday','Thursday','Friday',
        'Saturday']
        self.listBox = wx.ListBox(panel, choices=li)             #创建列表框
        self.Bind(wx.EVT_LISTBOX, self.OnClick, self.listBox)    #绑定事件处理函数

    def OnClick(self, event):
        #t = self.listBox.GetSelection()
        #s = self.listBox.GetString(t)
        s = self.listBox.GetStringSelection()
        wx.MessageBox(s)

    def OnButtonQuit(self, event):
        dlg=wx.MessageDialog(self,'Really Quit?','Caution',\
                             wx.CANCEL|wx.OK|wx.ICON_QUESTION)
        if dlg.ShowModal() ==wx.ID_OK:
            self.Destroy()
if __name__ =='__main__':
    app = wx.App()
    frame = ListBoxDemo(None)
    frame.Show()
    app.MainLoop()
```

运行结果如图 9-4 所示。

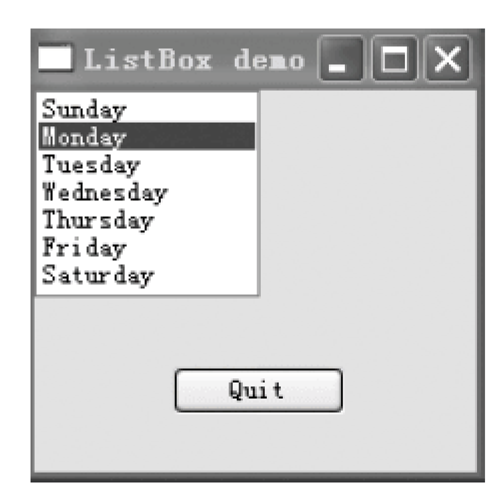


图 9-4　列表框用法

9.2.7　TreeCtrl

树形控件常用来显示有严格层次关系的数据，可以非常清晰地表述各元素之间的从属关系或层级关系，比如 Windows 资源管理器左侧窗口（见图 9-5）以及注册表编辑器（见图 9-6）。

树形控件的构造函数语法如下：

```
__init__(self, Window parent, int id=-1, Point pos=DefaultPosition, Size size=DefaultSize,
long style = TR_DEFAULT_STYLE, Validator validator = DefaultValidator, String name =
TreeCtrlNameStr)
```

由于大多数参数均有默认值，使用下面的代码就可以创建一个简单的树形控件：

```
tree = wx.TreeCtrl(panel)
```

树形控件的常用方法和事件分别如表 9-4 和表 9-5 所示。

表 9-4　树形控件的常用方法

方　法	说　明
root = tree.AddRoot(string)	增加根节点，返回根节点 ID
child=tree.AppendItem(item, string)	为指定节点增加下级节点，返回新节点 ID
SetItemText(item, string)	设置节点文本

续表

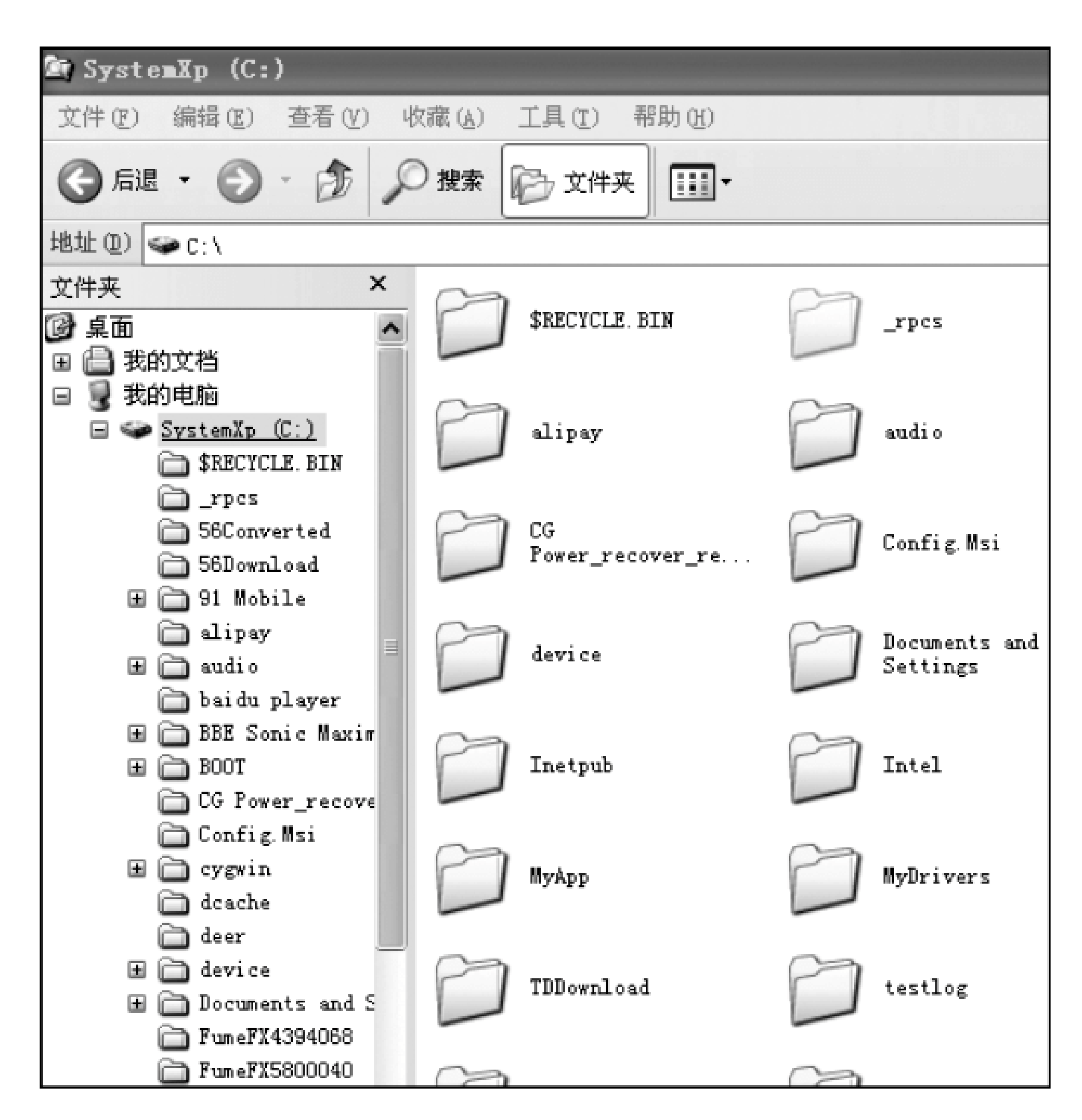


图 9-5 资源管理器界面

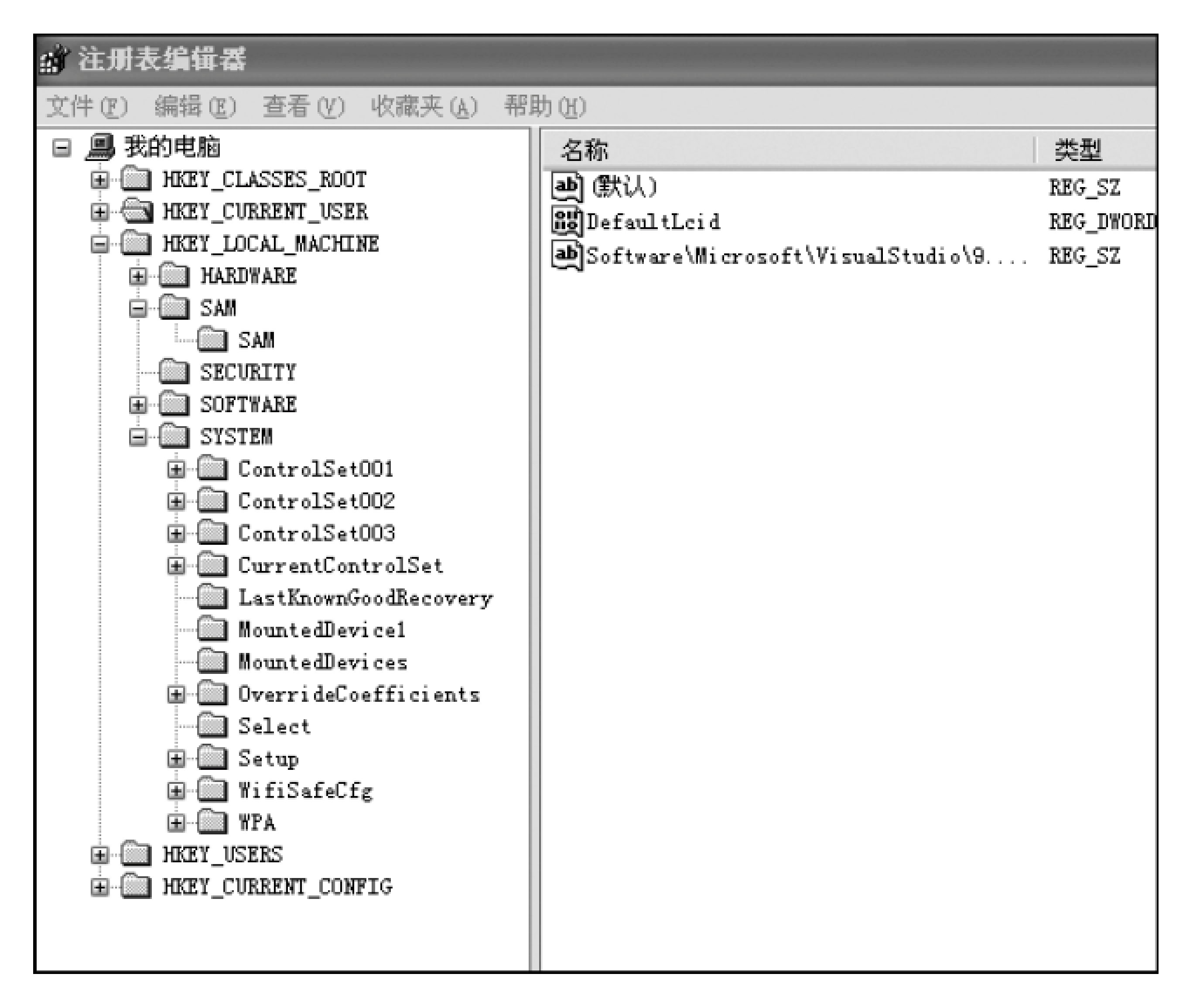


图 9-6 注册表编辑器界面

方 法	说 明
GetItemText()	返回节点文本
SetItemPyData(item，ojb)	设置节点数据
GetItemPyData(item)	返回指定节点的数据

方　　法	说　　明
Expand(item)	展开指定节点,但不展开下级节点
ExpandAll()	展开所有节点
Collapse(item)	收起指定节点
CollapseAndReset()	收起指定节点并删除其下级节点
GetRootItem()	返回根节点 ID
(childID, cookie) = GetFirstChild(item)	返回指定节点的第一个子节点
flag = child.IsOk()	测试节点 ID 是否有效
(item, cookie) = GetNextChild(item, cookie)	返回同级的下一个节点
GetLastChild(item)	返回指定节点的最后一个子节点
GetPrevSibling(item)	返回同级的上一个节点
GetItemParent(item)	返回指定节点的父节点 ID
ItemHasChildren(item)	测试节点是否有下级节点
SetItemHasChildren(item, True)	将指定节点设置为有下级节点的状态
GetSelection()	返回单选树中当前被选中节点的 ID
GetSelections()	返回多选树中所有被选中节点 ID 的列表
SelectItem(item, True/False)	改变节点的选择状态
IsSelected(item)	测试节点是否被选中
Delete(item)	删除指定 ID 的节点
DeleteAllItems()	删除所有节点
DeleteChildren(item)	删除指定 ID 的节点所有下级节点
InsertItem(parent, idPrevious, text)	在指定节点后面插入节点
InsertItemBefore(parent, index, text)	在指定位置之前插入节点

表 9-5　树形控件的常用事件

事　　件	说　　明
wx.EVT_TREE_SEL_CHANGING	控件发生选择变化之前触发该事件
wx.EVT_TREE_SEL_CHANGED	控件发生选择变化之后触发该事件
wx.EVT_TREE_ITEM_COLLAPSING	收起一个节点之前触发该事件
wx.EVT_TREE_ITEM_COLLAPSED	收起一个节点之后触发该事件
wx.EVT_TREE_ITEM_EXPANDING	展开一个节点之前触发该事件
wx.EVT_TREE_ITEM_EXPANDED	展开一个节点之后触发该事件

下面的代码演示了树形控件的用法,这个简单的示例演示了增加根节点、增加子节点、删除节点等功能。

续表

```
import wx

class TreeCtrlFrame(wx.Frame):
    def __init__(self, superion):
        wx.Frame.__init__(self, parent=superion, title='TreeCtrl demo', size=(300,400))
        panel = wx.Panel(self)
        self.tree = wx.TreeCtrl(parent=panel, pos=(5,5), size=(120,200))
        self.inputString = wx.TextCtrl(parent=panel, pos=(150,10))
        self.buttonAddChild = wx.Button(parent=panel, label='AddChild', pos=(150,90))
        self.Bind(wx.EVT_BUTTON, self.OnButtonAddChild, self.buttonAddChild)
        self.buttonDeleteNode = wx.Button(parent=panel, label='DeleteNode', pos=(150,120))
        self.Bind(wx.EVT_BUTTON, self.OnButtonDeleteNode, self.buttonDeleteNode)
        self.buttonAddRoot = wx.Button(parent=panel, label='AddRoot', pos=(150,150))
        self.Bind(wx.EVT_BUTTON, self.OnButtonAddRoot, self.buttonAddRoot)

    def OnButtonAddChild(self, event):
        itemSelected = self.tree.GetSelection()
        if not itemSelected:
            wx.MessageBox('Select a Node first.')
            return
        itemString = self.inputString.GetValue()
        self.tree.AppendItem(itemSelected, itemString)

    def OnButtonDeleteNode(self, event):
        itemSelected = self.tree.GetSelection()
        if not itemSelected:
            wx.MessageBox('Select a Node first.')
            return
        self.tree.Delete(itemSelected)
    def OnButtonAddRoot(self, event):
        rootItem = self.tree.GetRootItem()
        if rootItem:
            wx.MessageBox('The tree already has a root.')
        else:
            itemString = self.inputString.GetValue()
            self.tree.AddRoot(itemString)

if __name__ == '__main__':
    app = wx.App()
    frame = TreeCtrlFrame(None)
    frame.Show()
    app.MainLoop()
```

运行结果如图 9-7 所示，界面中的文本框用来输入节点显示的文本，首先单击按钮

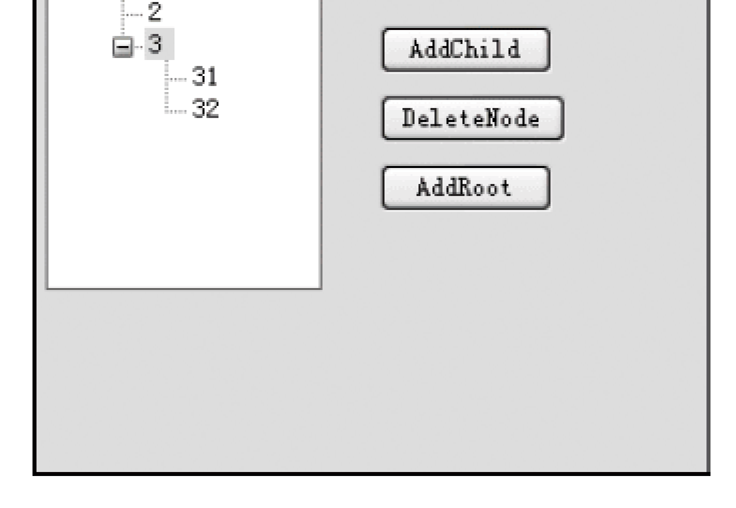


图 9-7 树形控件用法

AddRoot 插入根节点，然后单击选中根节点，再单击按钮 AddChild 插入子节点，最后单击选中某个子节点再单击按钮 AddChild 为其插入子节点。如果单击选中某个子节点后单击按钮 DeleteNode 则可以删除该节点。

9.3 Boa-constructor

Boa-constructor 是基于 wxPython 开发的一款优秀的界面设计软件，同时也是一款不错的 Python 集成开发环境，使用 Boa-constructor 设计界面时可通过鼠标调整控件的大小和位置，而不用像 IDLE 那样完全依靠手写代码来创建和控制界面。

登录 http://sourceforge.net/projects/boa-constructor 下载 Boa-constructor 安装程序后，安装过程如同其他 Windows 应用程序一样，无须多做解释。启动 boa-constructor 时需要首先打开 IDLE，然后在 IDLE 中打开并运行 C:\Python27\Lib\site-packages\boa-constructor\Boa.py 文件，或者直接双击运行该文件，当然也可以在桌面或者您更习惯的位置创建快捷方式。启动之后的界面基本上一目了然，本书不再详述，请大家自行查阅相关资料。

本章知识精要

(1) wxPython 是跨平台的 GUI 模块，除此之外，还可以使用 Python 内置的 Tkinter 以及基于 Java 的 Jython 和支持.NET 的 IronPython 等开发环境来支持 GUI 编程。

(2) 框架类 Frame 是包含标题栏、菜单、按钮、单选按钮、文本框和组合框等其他控件的容器。

(3) 静态文本控件 StaticText 一般不用来接受用户的单击、双击等交互操作。

(4) 按钮控件 Button 上显示的文本可以通过 SetLabelText()方法动态改变，结合获取文本的 GetLabelText()方法可以让一个按钮实现多个功能。

(5) 下拉菜单和弹出式菜单的菜单项事件处理函数绑定方式是一样的。

(6) 对话框需要首先创建，然后显示对话框，最后再获取对话框的返回值。

(7) 同一组中的多个单选按钮控件的选择是互斥的，而复选框控件的选择不是互斥的。

(8) 树形控件常用来显示有严格层次关系或从属关系的数据。

习　　题

1. 设计一个窗体，并放置一个按钮，单击按钮后弹出“颜色”对话框，关闭“颜色”对话框后提示选中的颜色。

2. 设计一个窗体，并放置一个按钮，按钮默认文本为“开始”，单击按钮后文本变为“结束”，再次单击后变为“开始”，循环切换。

3. 设计一个窗体，模拟 QQ 登录界面，当用户输入号码 123456 和密码 654321 时提示

正确,否则提示错误。

第 10 章　网络程序设计

Socket 是计算机之间进行网络通信的一套程序接口，最初由 Berkeley 大学研发，目前已经成为网络编程的标准，可以实现跨平台的数据传输。Socket 是网络通信的基础，相当于在发送端和接收端之间建立了一个管道来实现数据和命令的相互传递。Python 提供了 socket 模块，对 Socket 进行了二次封装，支持 Socket 接口的访问，大幅度简化了程序的开发步骤，提高了开发效率。除此之外，Python 还提供了 urllib 等大量模块可以对网页内容进行读取和处理，在此基础上结合多线程编程以及其他有关模块可以快速开发网页爬虫之类的应用。可以使用 Python 语言编写 CGI 程序，也可以把 Python 代码嵌入到网页中运行，而借助于 web2py 框架，则可以快速开发网站应用。本章将依次介绍上述几个方面的应用开发。

10.1　计算机网络基础知识

为了更好地理解本章后面的内容，本节首先简要介绍一下计算机网络的有关概念，如果您感兴趣的话可以参考《计算机网络》之类的书籍以获取更详细的知识。

(1) 网络体系结构。目前较为主流的网络体系结构是 ISO/OSI 参考模型和 TCP/IP 协议族。这两种体系结构都采用了分层设计和实现的方式。例如，ISO/OSI 参考模型从上而下划分为应用层、表示层、会话层、传输层、网络层、数据链路层和物理层，而 TCP/IP 则将网络划分为应用层、传输层、网络层和链路层。分层设计的好处是，各层可以独立设计和实现，只要保证相邻层之间的调用规范和调用接口不变，就可以方便、灵活地改变某层的内部实现以进行优化或完成其他需求。

(2) 网络协议。网络协议是计算机网络中进行数据交换而建立的规则、标准或约定的集合。网络协议的三要素分别为语法、语义和时序。简单地讲，可以这么理解，语义表示要做什么，语法表示要怎么做，时序规定了各种事件出现的顺序。

① 语法。语法规定了用户数据与控制信息的结构与格式，以及数据和控制信息出现的顺序。

② 语义。语义用来解释控制信息每个部分的意义，规定了需要发出何种控制信息，以及需要完成的动作和做出什么样的响应。

③ 时序。时序是对事件发生顺序的详细说明，也可称为"同步"。

(3) 应用层协议。应用层协议直接与最终用户进行交互，定义了运行在不同终端系统上的应用程序进程如何相互传递报文。下面简单列出 6 种常见的应用层协议。

① DNS。域名服务，用来实现域名与 IP 地址的转换。

② FTP。文件传输协议，可以通过网络在不同平台之间实现文件的传输。

③ HTTP。超文本传输协议。

④ SMTP。简单邮件传输协议。

⑤ ARP。地址解析协议，实现 IP 地址与 MAC 地址的转换。

⑥ TELNET。远程登录协议。

(4) 传输层协议。在传输层主要运行着 TCP 和 UDP 两个协议,其中 TCP 是面向连接的、具有质量保证的可靠传输协议,但开销较大;UDP 是尽最大能力传输的无连接协议,开销小,常用于视频在线点播(Video On Demand, VOD)之类的应用。TCP 和 UDP 并没有优劣之分,仅仅是适用场合有所不同。在传输层,使用端口号来标识和区分具体的应用层进程,每当创建一个应用层网络进程时系统就会自动分配一个端口号与之关联,是实现网络上端到端通信的重要基础。

(5) IP 地址。IP 运行于网络体系结构的网络层,是网络互连的重要基础。IP 地址(32 位或 128 位二进制数)用来标识网络上的主机,在公开网络上或同一个局域网内部,每台主机都必须使用不同的 IP 地址;而由于网络地址转换(Network Address Translation, NAT)和代理服务器等技术的广泛应用,不同内网之间的主机 IP 地址可以相同并且可以互不影响地正常工作。IP 地址与端口号共同来标识网络上特定主机上的特定应用进程,俗称 Socket。

(6) MAC 地址。MAC 地址也称为网卡地址或物理地址,是一个 48 位的二进制数,用来标识不同的网卡物理地址。本机的 IP 地址和 MAC 地址可以在命令提示符窗口中使用 ipconfig/all 命令查看,请参考后面的图 10-1。

10.2 UDP 和 TCP 编程

如前所述,UDP 和 TCP 是网络体系结构的运输层(也称为传输层)运行的两大重要协议,其中 TCP 适用于对效率要求相对低而对准确性要求相对高的场合,例如文件传输、电子邮件等;而 UDP 适用于对效率要求相对高,对准确性要求相对低的场合,例如视频在线点播、网络语音通话等。在 Python 中,主要使用 socket 模块来支持 TCP 和 UDP 编程。

10.2.1 UDP 编程

UDP 属于无连接协议,在 UDP 编程时不需要首先建立连接,而是直接向接收方发送信息。UDP 编程经常用到的 socket 模块方法有 3 个。

(1) socket([family[,type[,proto]]]):创建一个 Socket 对象,其中 family 为 socket.AF_INET 表示 IPV4,socket.AF_INET6 表示 IPV6;type 为 SOCK_STREAM 表示 TCP,SOCK_DGRAM 表示 UDP。

(2) sendto(string,address):把 string 指定的内容发送给 address 指定的地址,其中 address 是一个包含接收方主机 IP 地址和应用进程端口号的元组,格式为(IP 地址,端口号)。

(3) recvfrom(bufsize[,flags]):接收数据。

下面通过一个示例来简单了解如何使用 UDP 进行网络通信。

【例 10-1】 编写 UDP 通信程序,发送端发送一个字符串“Hello world!”。假设接收端在计算机的 5005 端口进行接收,并显示接收内容。

接收端代码:

```
import socket
s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(("", 5005))                          #空字符串表示本机任何可用 IP 地址
```

```
data, addr=s.recvfrom(1024)            #缓冲区大小为 1024B
print ' received message:%s' %data     #显示接收到的内容
s.close()
```

发送端代码：

```
import socket
s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.sendto("Hello, world!" , ("10.20.52.248" ,5005))  #假设 10.20.52.248 是接收端主机的 IP 地址
s.close()
```

需要说明的是，在上面的发送端程序中假设接收端主机 IP 地址为 10.20.52.248，这很可能与您的计算机配置不一样，从而导致运行结果与图 10-2 不同。您需要使用命令 ipconfig/all 查看本机 IP 地址，如图 10-1 所示，然后对发送端代码中的 IP 地址进行相应修改。



图 10-1　查看本机 IP 地址

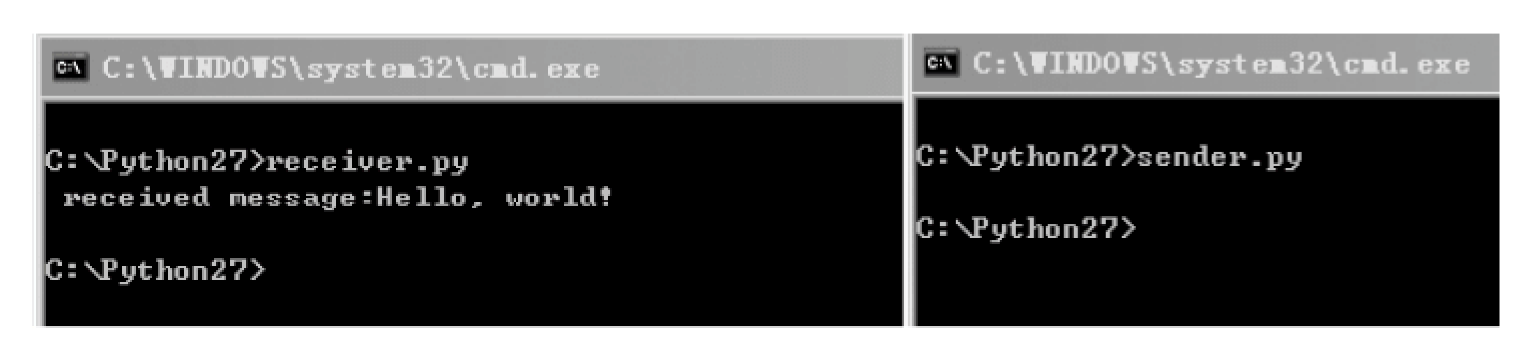
将上面的代码分别保存为 receiver.py 和 sender.py，然后需要首先在一个命令提示符界面中运行接收端程序，这时会发现接收端程序处于阻塞状态。接下来启动另一个命令提示符窗口并运行发送端程序，此时会看到接收端程序继续运行并显示接收到的内容，如图 10-2 所示。

10.2.2　TCP 编程

TCP 一般用于要求可靠数据传输的场合。编写 TCP 程序时经常需要用到的 socket 模块的方法主要有 6 个。



(a) 运行接收端程序,处于阻塞状态



(b) 运行发送端程序,接收端程序继续运行并显示接收的内容

图 10-2　UDP 通信程序运行结果

(1) connect(address):连接远程计算机。

(2) send(bytes[,flags]):发送数据。

(3) recv(bufsize[,flags]):接收数据。

(4) bind(address):绑定地址。

(5) listen(backlog):开始监听,等待客户端连接。

(6) accept():响应客户端的请求。

下面通过一个示例来演示如何使用 TCP 进行通信。

【例 10-2】 TCP 通信程序。

使用 TCP 进行通信需要首先在客户端和服务端之间建立连接,并且要在通信结束后关闭连接以释放资源。

客户端代码文件 client.py:

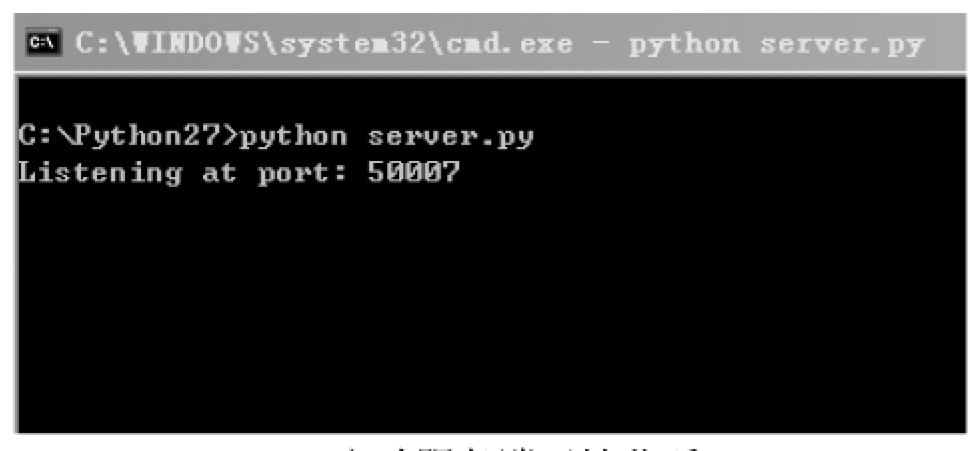
```
import socket
HOST ='127.0.0.1'                    #服务端主机 IP 地址
PORT =50007                          #服务端主机应用进程端口号
s =socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))              #建立连接
s.sendall(b'Hello, world')           #发送数据
data =s.recv(1024)                   #从服务端接收数据
s.close()                            #关闭连接
print 'Received', repr(data)
```

服务端代码文件 server.py:

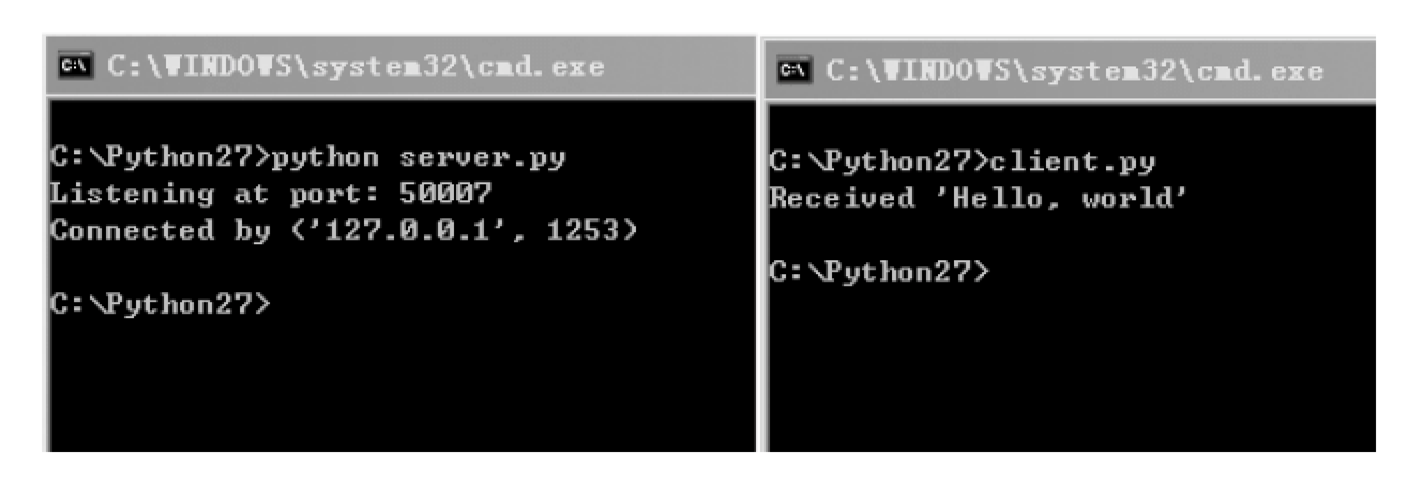
```
import socket
HOST =''                             #本机所有可用 IP 地址
PORT =50007
s =socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))                 #绑定 Socket
s.listen(1)                          #开始监听
print 'Listening at port:',PORT
```

```
conn, addr = s.accept()
print 'Connected by', addr
while True:
    data = conn.recv(1024)
    if not data:
        break
    conn.sendall(data)
conn.close()
```

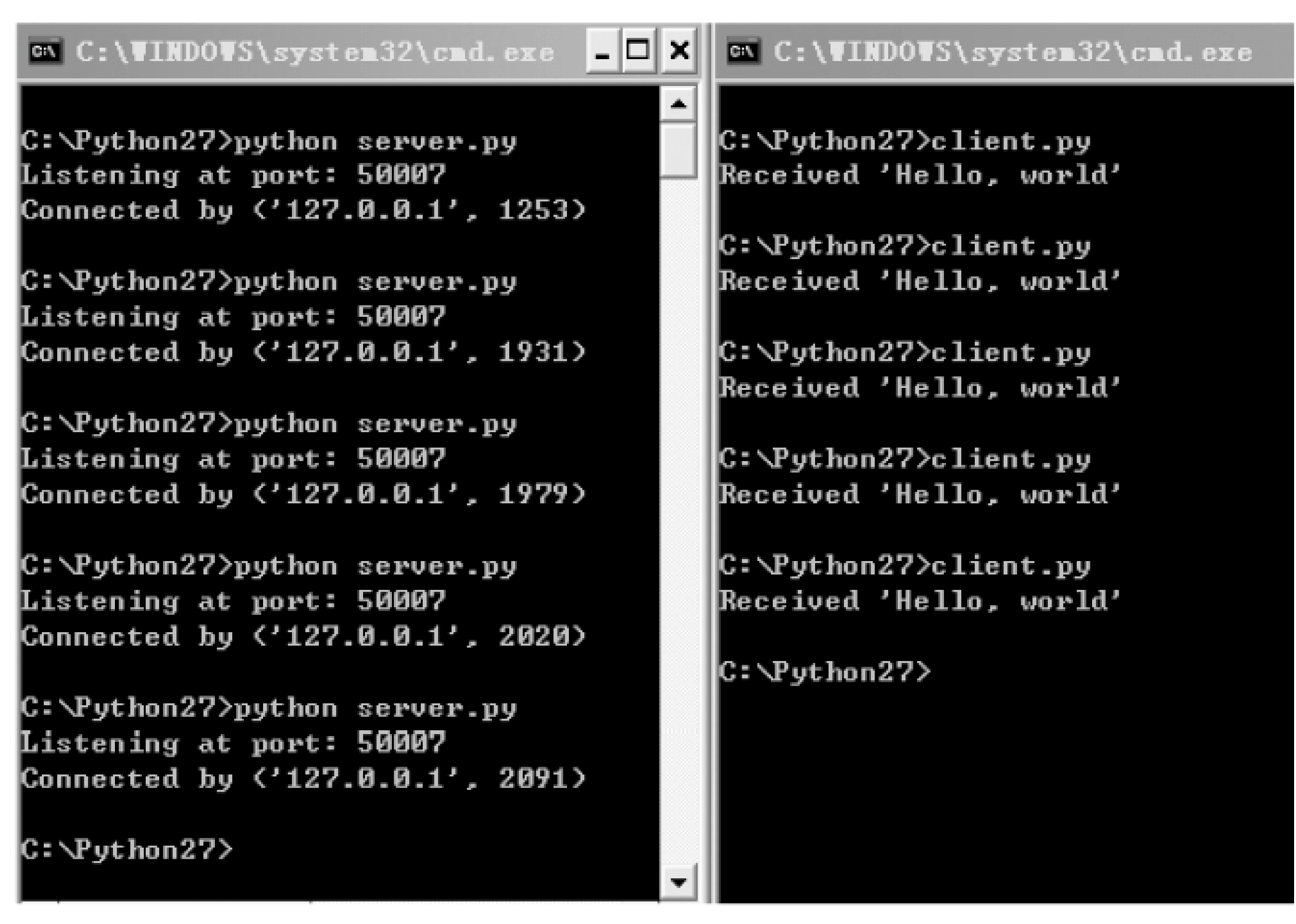
将上面两段代码分别保存为 client.py 和 server.py 文件，启动一个命令提示符窗口，运行服务端程序开始监听，如图 10-3(a)所示；然后再启动一个命令提示符窗口并运行客户端程序，向服务端发送数据，服务端程序收到数据后向客户端回复数据，如图 10-3(b)所示；多次运行服务端程序和客户端程序可以发现，服务端程序总是在预先设置的固定端口监听，而客户端每次连接时所使用的端口却是变化的，如图 10-3(c)所示。



(a) 启动服务端开始监听



(b) 运行客户端程序



(c) 多次运行服务端和客户端程序

图 10-3 TCP 通信程序运行结果

10.3 简单嗅探器实现

嗅探器程序可以检测和监控本机所在局域网内的网络流量和所有数据包，对于网络管理来说具有重要作用。为了实现网络流量嗅探，需要将网卡设置为混杂模式，下面程序的运行需要获得管理员权限。

```
import socket
#the public network interface
HOST = socket.gethostbyname(socket.gethostname())
#create a raw socket and bind it to the public interface
s = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_IP)
s.bind((HOST, 0))
#include IP headers
s.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
#receive all packages
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
#receive a package
print(s.recvfrom(65565))
#disabled promiscuous mode
s.ioctl(socket.SIO_RCVALL, socket.RCVALL_OFF)
```

10.4 网页内容读取

在 Python 2 中提供了 urllib 和 urllib2 两个模块用来支持网页内容读取，在 Python 3 中对这两个模块进行了整合，只提供 urllib 一个模块。Python 2 中的 urllib 模块在 Python 3 中被分成了 urllib.request、urllib.parse 和 urllib.error 三部分，目前只支持 HTTP (versions 0.9 and versions 1.0)、FTP 和 local files 3 种协议。urllib2 模块在 Python 3 中则被合并到了 urllib.request 和 urllib.error 中。

10.4.1 urllib

该模块提供了大量方法，具体列表可以导入 urllib 模块之后使用 dir()函数来查看，并使用 help()函数来获取特定函数的详细帮助文档。本节主要通过几个示例来简单演示该模块的用法。

下面的 Python 2.7.8 代码用来输出指定网页的内容：

```
>>>import urllib
>>>f = urllib.urlopen('http://www.python.org')
>>>print f.read()
>>>f.close()
```

在 Python 3.4.2 中上面的代码应替换为下面的代码，后面的几个示例与此类似，请自行查阅帮助文档进行正确的替换。

```
>>> import urllib.request
>>> dir(urllib.request)
>>> fp = urllib.request.urlopen('http://www.python.org')
>>> dir(fp)
>>> print(fp.read(100))
>>> fp.close()
```

下面的 Python 2.7.8 代码使用 GET 方法提交参数并访问指定页面：

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query?%s" %params)
>>> print f.read()
```

下面的 Python 2.7.8 代码使用 POST 方法提交参数并访问指定页面：

```
>>> import urllib
>>> params = urllib.urlencode({'spam': 1, 'eggs': 2, 'bacon': 0})
>>> f = urllib.urlopen("http://www.musi-cal.com/cgi-bin/query", params)
>>> print f.read()
```

下面的 Python 2.7.8 代码使用 HTTP 代理访问指定网页：

```
>>> import urllib
>>> proxies = {'http': 'http://proxy.example.com:8080/'}
>>> opener = urllib.FancyURLopener(proxies)
>>> f = opener.open("http://www.python.org")
>>> f.read()
```

10.4.2 其他可能用到的模块

除了 urllib 模块之外，在实际开发中可能还需要结合其他有关模块来实现特定的需求，下面列出了其中 6 个。

（1）cookielib：提供了可存储和操作 cookie 的对象与方法，以便于与 urllib 模块配合使用来访问 Internet 资源。

（2）httplib：是相对底层的 HTTP 请求处理模块，不如 urllib 模块封装的层次高，但能够更加灵活地对通信进行控制。

（3）beautifulSoup：使用 Python 实现的一个 HTML/XML 的解析器，可以很好地处理不规范标记并生成剖析树(parse tree)。

（4）Scrapy：使用 Python 实现的一个快速、高层的屏幕抓取和 Web 抓取框架，用于抓取 Web 站点并从页面中提取结构化的数据。Scrapy 用途非常广泛，可以用于数据挖掘、监测和自动化测试等。

（5）json、BeJson：轻量级数据交换格式，易于阅读和编写，同时也易于机器解析和生成，使用该格式可以提高网络传输速度。

（6）rsa：密码模块，可以用来结合 urllib 模块实现某些网站或论坛的模拟登录。

10.5 使用 Python 开发网站

Python 是一门脚本语言，完全可以像 PHP、VBScript 等脚本语言一样用来开发网页以及 CGI 程序。既可以直接编写 Python 脚本程序来生成网页，也可以把 Python 程序嵌入.asp 文件，无论使用哪种方式，都可以使用服务器上已安装的所有 Python 扩展模块，同时也要遵守缩进以及其他格式要求，如同编写 Python 程序一样。在 Windows 平台上，为了能让 IIS 运行 Python 程序，需要完成以下几步设置。

(1) 设置 IIS 中网站属性，重点是 TCP 端口号的设置，这里设置为 8080，如图 10-4 所示。



图 10-4 "网站"选项卡

(2) 创建网站或虚拟目录，设置别名，这里以 Python 为例，如图 10-5 所示。



图 10-5 创建虚拟目录

(3) 接下来为刚才创建的网站或虚拟目录设置目录，即包含 Python 程序或内嵌 Python 代码的.asp 文件的目录，如图 10-6 所示。

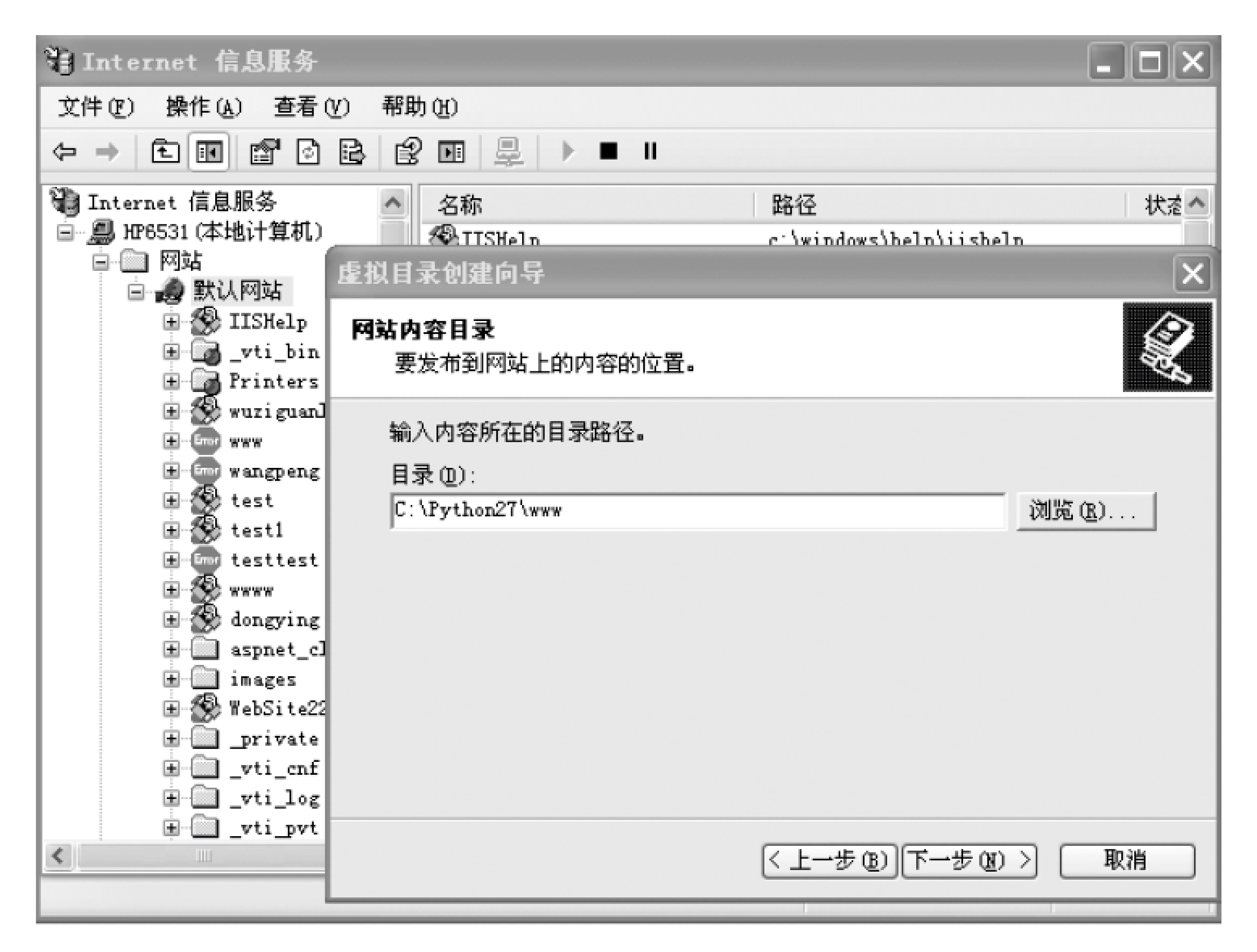


图 10-6　设置目录

(4) 打开网站属性界面，设置脚本访问权限，如图 10-7 所示。



图 10-7　设置权限

最后单击“配置”按钮，在弹出来的“映射”选项卡中单击“添加”按钮，然后填写可执行文件以及扩展名来设置 Python 程序的映射关系，如图 10-8 所示。

(5) 编写 Python 程序 index.py，并保存在上面设置的目录中。

```
print
print 'Status: 200 OK'
print 'Content-type: text/html'
```



图 10-8 设置 Python 程序映射关系

```
print
print '<html><head><title>Python Sample CGI</title></head>'
print '<body>'
print '<h1>This is a header</h1>'
print '<p>note:this is only a test.'
print '<br>'
print '</body>'
```

(6) 使用浏览器访问刚才创建的网站,如图 10-9 所示。



图 10-9 使用 Python 编写 CGI 程序

下面的代码演示了在.asp 文件中内嵌 Python 代码的用法:

```
<%@LANGUAGE=Python%>
<html>
<head></head>
<body>
<h1>Python Test</h1>
<%
import math
s =math.sqrt(9)
Response.Write('Python Test<br>')
Response.Write(str(s))
```

```
Response.write('<h3>Smaller heading</hr>')
%>
</body>
</html>
```

将上面的代码保存为 index1.asp 文件，并放置于前面在 IIS 中设置的目录 C:\Python27\www 中，通过浏览器访问该网页，结果如图 10-10 所示。



图 10-10　内嵌 Python 代码的.asp 网页

10.6　使用 web2py 框架开发网站

Zope2、Web.py、Pyramid、CubicWeb、Django 和 web2py 是目前比较流行的支持 Python 的网站开发框架。尽管每个框架都有自己的特色和独到之处，但其中用户推荐度较高的当属 web2py，该框架中集成了用户认证、数据库操作、模板系统和 Form 表单等大量功能组件，能够完成开发中的常用功能。开发者通过组合不同的功能组件，再加上自己实现的业务逻辑，就像搭积木一样来快速开发 Web 应用。

本节以 web2py 框架为例介绍网站开发流程。web2py 框架使用 MVC 模式实现网站开发，即 Model-View-Controller 模式。其中 M 指模型，即存储在数据库中的待处理数据；V 指视图，用来决定数据的显示形式；C 指控制，即负责处理用户请求，根据特定的业务逻辑对模型中的数据进行修改并将新的结果视图返回给最终用户。使用 web2py 框架开发 Web 应用的流程：首先定义模型，然后编写控制逻辑，最后实现视图，将数据以特定的形式展示给最终用户。

您可以登录 web2py 框架官方主页(http://web2py.com)下载适合您的压缩包，解压缩后找到并执行 web2py.exe 文件，会看到一个黑色的命令提示符窗口和 web2py 框架的主窗口，选择服务器的 IP 地址、设置服务器端口号和密码之后，单击 start server 按钮即可启动 web2py 框架并打开 web2py 的欢迎界面，单击页面右侧的 administrative interface 进入管理员页面，分别如图 10-11 和图 10-12 所示。

在 web2py 中每一个网站都是一个应用或者 APP，默认有 admin、examples 和 welcome 3 个应用，如图 10-12 所示。如果需要创建自己的应用，可以在图 10-12 中页面右侧输入要创建的应用名称然后单击 create 按钮，然后即可自动跳转到相应应用的设计与开发界面。下面我们通过一个例子来演示使用 web2py 开发 Web 应用的流程，要求网站运行后接收用

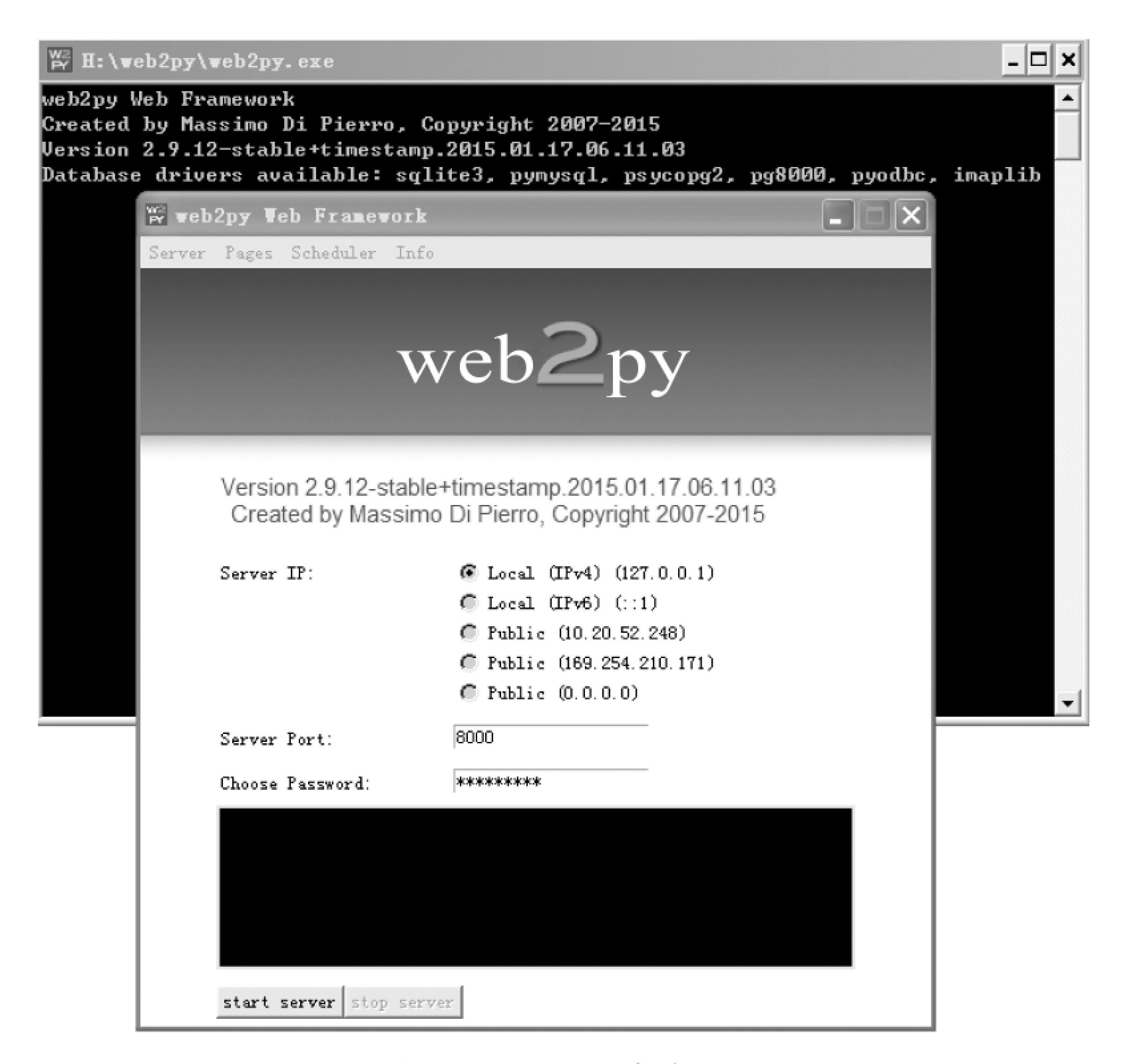


图 10-11　web2py 启动界面

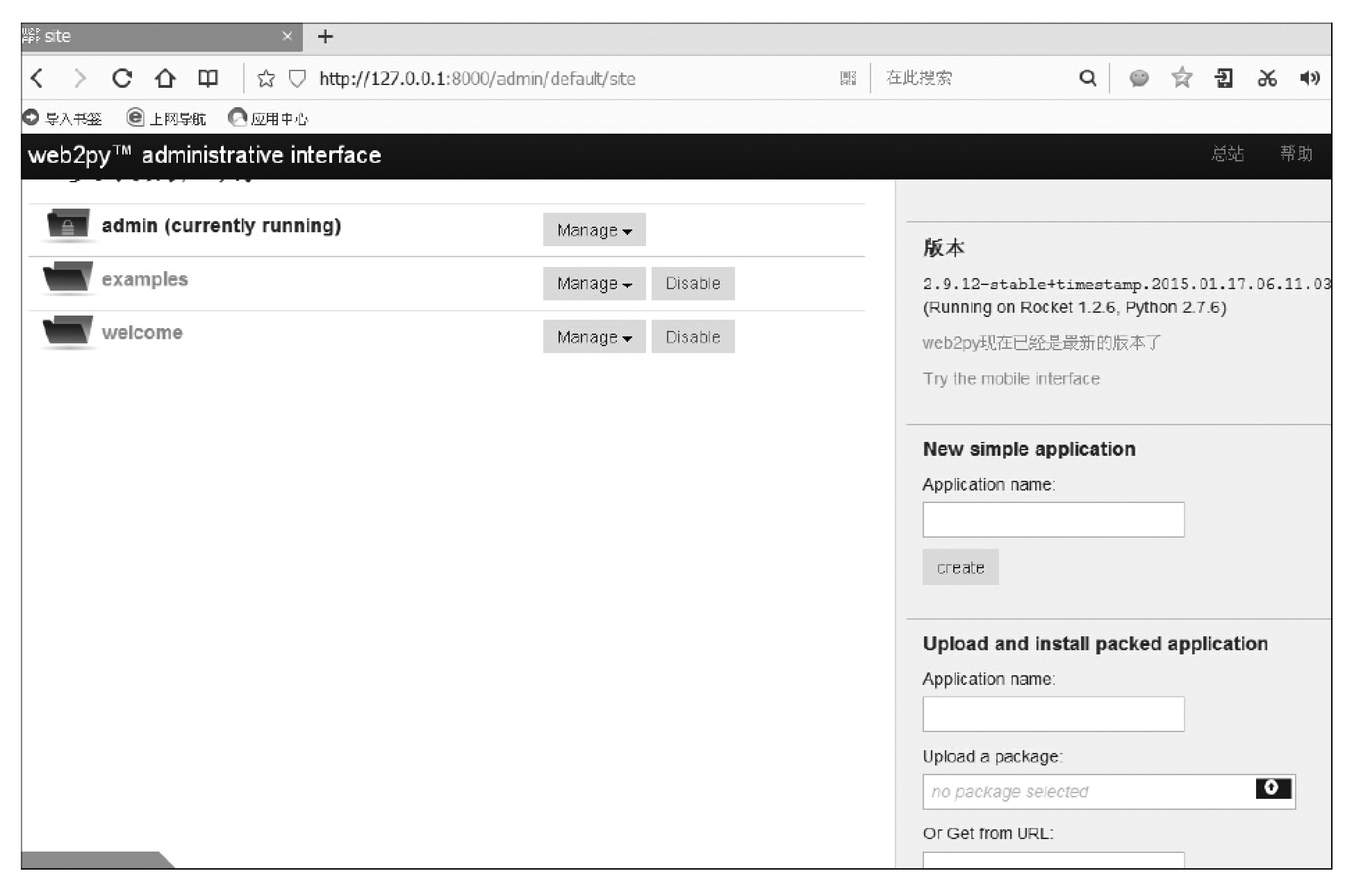


图 10-12　web2py 管理员页面

户输入的整数，单击按钮以后判断该整数是否为素数，并给出结果。更多详细资料请登录 web2py 官方网站进行查阅。

（1）首先在 web2py 管理员界面中右侧文本框中输入 IsPrime，然后单击 create 按钮，进入 IsPrime 应用的设计开发界面，如图 10-13 所示。



图 10-13　web2py 的应用设计开发界面

（2）单击控制器一栏下方的 create 按钮，然后输入 IsPrime 并再次单击下面的 create 按钮，即可看到新创建的文件 IsPrime. py，如图 10-14 所示。

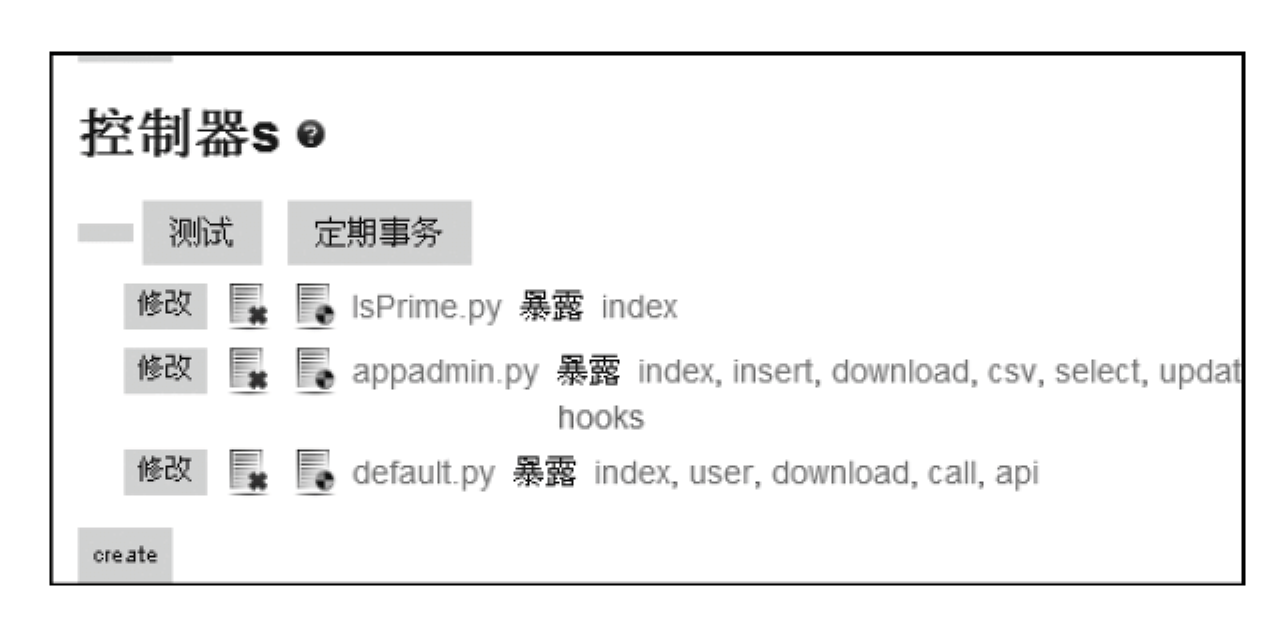


图 10-14　创建控制器文件

修改 IsPrime. py 文件内容，编写下面的代码：

```
#-*-coding: utf-8 -*-
def index():
    form = FORM(INPUT(_name='number', requires=IS_NOT_EMPTY()),\
                INPUT(_type='submit'))
    if form.process().accepted:
        num = int(form.vars.number)
        for i in range(2,num):
            if num%i ==0:
                session.r = 'No'            #设置会话变量
                redirect(URL('result'))     #重定向至 result.html
        else:
            session.r = 'Yes'
            redirect(URL('result'))
    return dict(form=form)
def result():
    r = session.r or redirect(URL('index'))
    return dict(r=r)
```

(3) 在视图下面使用 create 按钮分别创建 index 和 result 两个视图文件，内容如下，其中两对大括号"{{ }}"中是 Python 代码。

index. html 文件内容：

```
{{extend 'layout.html}}
{{=form}}
```

result. html 文件内容：

```
{{extend 'layout.html'}}
<h1>{{=session.r or 'Unknown'}}</h1>
```

然后在地址栏中输入地址 http://127. 0. 0. 1:8000/IsPrime/isprime/index，在文本框内输入一个整数，单击旁边的按钮，页面显示该整数是否为素数，如图 10-15 和图 10-16 所示。

在 web2py 解压缩目录的 applications 子目录中可以找到新建的 IsPrime 应用的所有文件，在其中的 views 子目录中有相应的视图文件，其中 layout. html 文件是 web2py 自带的页面样式，可以根据自己喜欢的风格和样式进行修改。

最后再给出几个简单的示例，可以通过 web2py 框架创建应用并运行。

1. 显示字符串

控制器代码(template_examples. py)：

```
def test_for():
    return dict()
```

视图代码(template_examples/test_for. html)：

```
{{extend 'layout.html'}}
```



图 10-15　index 页面



图 10-16　result 页面

```
<h1>For loop</h1>
{{for number in ['one','two','three']:}}
<h2>{{=number.capitalize()}}</h2>
{{pass}}
```

2．以超链接方式显示指定网址

控制器代码(template_examples. py)：

```
def test_def():
    return dict()
```

视图代码(template_examples/test_def. html)：

```
{{extend 'layout.html'}}
{{def itemlink(name):}}<li>{{=A(name,_href=name)}}</li>{{return}}
```

```
<ul>
{{itemlink('http://www.google.com')}}
{{itemlink('http://www.yahoo.com')}}
{{itemlink('http://www.nyt.com')}}
</ul>
```

3. 显示变量值

控制器代码(template_examples.py)：

```
def variables():
    return dict(a=10, b=20)
```

视图代码(template_examples/variables.html)：

```
{{extend 'layout.html'}}
<h1>Your variables</h1>
<h2>a={{=a}}</h2>
<h2>a={{=b}}</h2>
```

本章知识精要

(1) IP 地址和端口号共同来标识网络上特定主机上的特定应用进程，称为 Socket。

(2) TCP 适用于对效率要求较低而对准确性要求较高的场合，是面向连接、具有服务质量保证的可靠传输协议，而 UDP 属于无连接协议，可能会发生丢包或其他错误，适用于视频在线点播、网络语音通信之类的场合。

(3) urllib 模块提供了大量对象和方法支持网页内容读取，结合密码模块以及多线程编程可以轻松实现网络爬虫程序。

(4) 既可以使用 Python 编写 CGI 程序来动态生成网页，也可以把 Python 程序嵌入 .asp 文件。

(5) 在 Windows 平台上使用 IIS 运行 Web 应用时需要配置映射关系来支持 Python 程序的解释和执行。

(6) web2py 框架集成了用户认证、数据库操作和模板系统等大量功能组件以支持 Web 开发。

习 题

1. 简单解释 TCP 和 UDP 的区别。

2. 同学之间合作编写 UDP 通信程序，分别编写发送端和接收端代码，发送端发送一个字符串“Hello world!”。假设接收端在计算机的 5000 端口进行接收，并显示接收内容。

3. 简单介绍 socket 模块中用于 TCP 编程的常用方法。

4. 编写代码读取搜狐网页首页内容。

5. 在自己的机器上配置 IIS 以支持 Python 脚本的运行，然后使用 Python 编写脚本，运行后在网页上显示“Hello world!”。

第 11 章　大数据处理

相信大家已经很明显地感觉到，这是一个信息量极度膨胀的时代，大数据的概念自从提出来以后，迅速渗透到各行各业。那么到底什么是大数据呢？历史上有个著名的故事叫“草船借箭”，故事的主人公诸葛亮对天象的观察实际上就是对风、云、温度、湿度、光照和所处节气等大量多元化的非结构数据进行综合分析，最终通过复杂的计算得出了正确的结论并为最终决策提供了有力支持，这可以看作是大数据的一个经典应用。下面引用网上一则小故事，也许对您理解大数据有所帮助。

某比萨店的电话铃响了，客服人员拿起电话。

客服：这是×××比萨店。您好，请问有什么需要我为您服务？

顾客：你好，我想要一份……

客服：先生，请先把您的会员卡号告诉我。

顾客：16846146***。

客服：陈先生，您好！您是住在泉州路一号 12 楼 1205 室，您家电话是 2646****，您公司电话是 4666****，您的手机是 1391234****。请问您想用哪一个电话付费？

顾客：你为什么知道我所有的电话号码？

客服：陈先生，因为我们联机到 CRM 系统。

顾客：我想要一个海鲜比萨……

客服：陈先生，海鲜比萨不适合您。

顾客：为什么？

客服：根据您的医疗记录，你的血压和胆固醇都偏高。

顾客：那你们有什么可以推荐的？

客服：您可以试试我们的低脂健康比萨。

顾客：你怎么知道我会喜欢吃这种的？

客服：您上星期一在中央图书馆借了一本《低脂健康食谱》。

顾客：好。那我要一个家庭特大号比萨，要付多少钱？

客服：99 元，这个足够您一家六口吃了。但您母亲应该少吃，她上个月刚刚做了心脏搭桥手术，还处在恢复期。

顾客：那可以刷卡吗？

客服：陈先生，对不起。请您付现款，因为您的信用卡已经刷爆了，您现在还欠银行 4807 元，而且还不包括房贷利息。

顾客：那我先去附近的提款机提款。

客服：陈先生，根据您的记录，您已经超过今日提款限额。

顾客：算了，你们直接把比萨送我家吧，家里有现金。你们多久会送到？

客服：大约 30 分钟。如果您不想等，可以自己骑车来。

顾客：为什么？

客服：根据我们CRM全球定位系统的车辆行驶自动跟踪系统记录。您登记有一辆车号为SB-748的摩托车，而目前您正在解放路东段华联商场右侧骑着这辆摩托车。

顾客当即晕倒……

现如今，大数据的应用比比皆是，为各行各业都带来了巨大商机，也提供了重要的决策支持，请看下面的例子。

(1) 洛杉矶警察局和加利福尼亚大学合作利用大数据预测犯罪的发生。

(2) Google流感趋势(Google Flu Trends)利用搜索关键词预测禽流感的散布。

(3) 统计学家内特·西尔弗(Nate Silver)利用大数据预测了2012年美国选举结果。

(4) 麻省理工学院利用手机定位数据和交通数据建立城市规划。

(5) 梅西百货的实时定价机制。根据需求和库存的情况，该公司基于SAS的系统对多达7300万种货品进行实时调价。

(6) Tipp24 AG针对欧洲博彩业构建的下注和预测平台。该公司用KXEN软件来分析数十亿计的交易以及客户的特性，然后通过预测模型对特定用户进行动态的营销活动。这项举措减少了90%的预测模型构建时间。SAP公司正在试图收购KXEN。SAP想通过这次收购来扭转其长久以来在预测分析方面的劣势。

(7) 沃尔玛的搜索。这家零售业寡头为其网站自行设计了最新的搜索引擎Polaris，利用语义数据进行文本分析、机器学习和同义词挖掘等。根据沃尔玛负责人的说法，语义搜索技术的运用使得在线购物的完成率提升了10%～15%。“对沃尔玛来说，这就意味着数十亿美元的金额。”

(8) 快餐业的视频分析。该公司通过视频分析等候队列的长度，然后自动变化电子菜单显示的内容。如果队列较长，则显示可以快速供给的食物；如果队列较短，则显示那些利润较高但准备时间相对长的食品。

目前在学术界公认的大数据四大特征如下。

(1) 数据量巨大。从TB级别跃升到PB级别甚至EB、ZB级别。

(2) 数据类型繁多。非结构化数据越来越多，例如网络日志、视频、图片和地理位置信息等，这对数据处理能力提出了更高的要求。

(3) 价值密度低。例如，在一个小时连续不间断的监控视频中，真正有用的数据很可能只有几秒钟。如何通过强大的机器和高效的算法更迅速地完成数据的价值“提纯”，成为目前大数据背景下亟待解决的难题和重要的研究热点之一。另外，数据的来源直接导致分析结果的准确性和真实性。若数据来源是完整的并且是真实的，最终的分析结果以及决定将更加准确。

(4) 要求处理速度快。根据IDC的“数字宇宙”的报告，预计到2020年，全球数据使用量将达到35.2ZB。可以说，在如此海量的数据面前，处理数据的效率就是企业的生命。

11.1 大数据框架

Amazon、Google等很多大公司都推出了自己的大数据框架和服务，这里简单介绍较为经典和流行的MapReduce、Hadoop和Spark，然后在11.2节以MapReduce为例介绍大数据处理的思路和具体应用。

1. MapReduce

分布式计算框架，可以将单个大型计算作业分配给多台计算机执行，可以在短时间内完成大量工作，尤其适合数值型和标称型数据，但需要对行业领域具有一定理解后重写算法来完成特定的业务处理要求。MapReduce 的名字由函数式编程中常用的 map 和 reduce 两个单词组成。MapReduce 在大量节点组成的集群上运行，工作流程：单个作业被分成很多小份，输入数据也被切片并分发到每个节点，每个节点只在本地数据上做运算，对应的运算代码称为 Mapper，这个过程即 map 阶段；每个 Mapper 的输出通过某种方式组合，根据需要可能再进行重新排序，排序后的结果再被切分成小份并分发到各个节点进行下一步处理，这个过程称为 reduce 阶段，对应的代码称为 Reducer。不同类型的作业可能需要不同数量的 Reducer，并且，在任何时候，每个 Mapper 或 Reducer 之间都不进行通信，每个节点只负责处理自己的事务，并且只在分配到本地的数据集上进行运算。

2. Hadoop

Hadoop 是 MapReduce 框架的一个免费开源实现，采用 Java 语言编写，支持在大量机器上分布式处理数据。除了分布式计算之外，Hadoop 还自带分布式文件系统，可以在上面运行多种不同语言编写的分布式程序。Hadoop 在可伸缩性、健壮性、计算性能和成本上具有无可替代的优势，事实上已成为当前互联网企业主流的大数据分析平台。

3. Spark

Spark 是一个针对超大数据集合的低延迟集群分布式计算系统，比 MapReduce 快 40 倍左右。Spark 是 Hadoop 的升级版本，兼容 Hadoop 的 API，能够读写 Hadoop 的 HDFS HBASE 顺序文件等，与之不同的是将结果保存在内存中。Hadoop 作为第一代产品使用了 HDFS，第二代加入了 Cache 来保存中间计算结果，第三代则是 Spark 倡导的流技术 Streaming。

11.2 MapReduce 编程案例

MapReduce 编程思路非常简单，首先对大数据进行分割，切分为一定大小的数据；然后将分割的数据交给多个 Mapper 函数进行处理，Mapper 函数处理后将产生一组规模较小的数据，多个规模较小的数据再提交给 Reducer 函数进行处理，得到一个更小规模的数据或最终结果。对于不同的具体应用，需要根据特定的要求来编写不同的 Mapper 和 Reducer 代码，并且可能会需要多次迭代来最终完成任务，如图 11-1 所示。

了解了基本原理以后，接下来我们通过一个例子来演示 MapReduce 的应用。Windows 系统的升级日志文件一般较大，现假设要求统计日志文件中与不同日期有关的记录条数。首先将大文件切分成多个小文件，然后对每个小文件进行 Map 处理，然后对得到的处理结果再进行 Reduce 处理，最终得到所需要的数据和结论。

1. 大文件切分(FileSplit.py)

```
import os
import os.path
import time
```

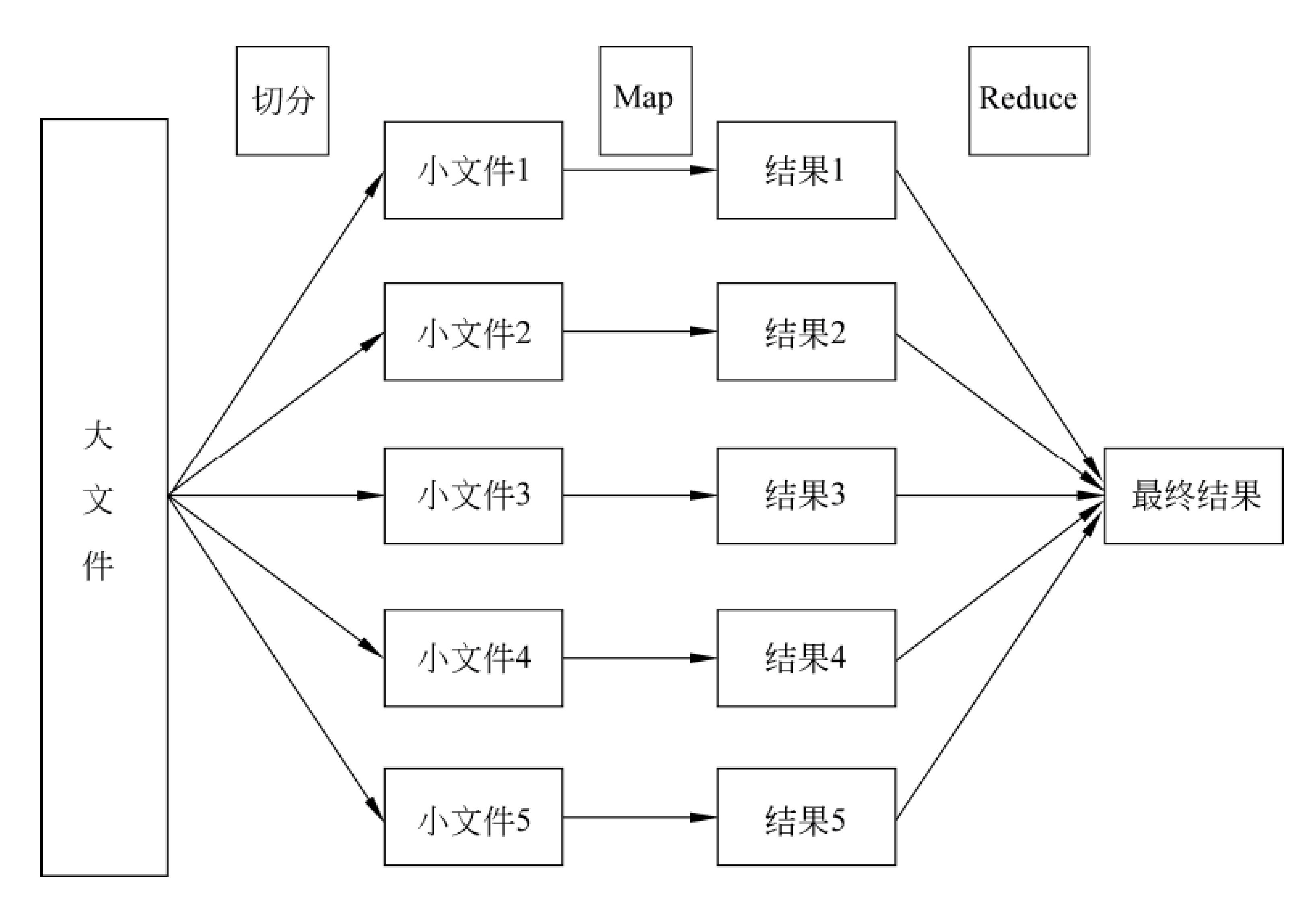


图 11-1 MapReduce 流程

```
def FileSplit(sourceFile, targetFolder):
    if not os.path.isfile(sourceFile):      #判断文件是否存在
        print sourceFile, ' does not exist.'
        return
    if not os.path.isdir(targetFolder):     #切分结果是多个小文件,存放到指定文件夹中
        os.mkdir(targetFolder)
    tempData = []
    number = 1000                            #每个小文件中的记录条数
    fileNum = 1                              #小文件序号
    with open(sourceFile, 'r') as srcFile:
        dataLine = srcFile.readline().strip()
        while dataLine:
            for i in range(number):
                tempData.append(dataLine)
                dataLine = srcFile.readline()
                if not dataLine:
                    break
            desFile = os.path.join(targetFolder, sourceFile[0:-4] + str(fileNum) + '.txt')
            with open(desFile, 'a+') as f:
                f.writelines(tempData)
            tempData = []
            fileNum = fileNum + 1

if __name__ == '__main__':
    #sourceFile = input('Input the source file to split:')
    #targetFolder = input('Input the target folder you want to place the split files:')
    sourceFile = 'test.txt'
    targetFolder = 'test'
```

```
    FileSplit(sourceFile, targetFolder)
```

2. Mapper 代码(Map. py)

```
import os
import re
import threading
import time

def Map(sourceFile):
    if not os.path.exists(sourceFile):
        print sourceFile, ' does not exist.'
        return
    pattern = re.compile(r'[0-9]{1,2}/[0-9]{1,2}/[0-9]{4}')              #使用正则表达式匹配日期
    result = {}
    with open(sourceFile, 'r') as srcFile:
        for dataLine in srcFile:
            r = pattern.findall(dataLine)
            if r:
                t = result.get(r[0], 0)
                t += 1
                result[r[0]] = t
    desFile = sourceFile[0:-4] + '_map.txt'                                #结果文件
    with open(desFile, 'a+') as fp:
        for k, v in result.items():
            fp.write(k + ':' + str(v) + '\n')

if __name__ == '__main__':
    desFolder = 'test'
    files = os.listdir(desFolder)

    #如果不使用多线程,可以直接这样写
    '''for f in files:
        Map(desFolder + '\\' + f)'''

    #使用多线程
    def Main(i):
        Map(desFolder + '\\' + files[i])
    fileNumber = len(files)
    for i in range(fileNumber):
        t = threading.Thread(target = Main, args = (i,))
        t.start()
```

3. Reducer 代码(Reduce. py)

```
import os

def Reduce(sourceFolder, targetFile):
```

```
    if not os.path.isdir(sourceFolder):
    print sourceFolder, ' does not exist.'
    return
result = {}
#使用列表推导式来获取文件夹中的 Mapper 结果文件
allFiles = [sourceFolder+'\\'+f for f in os.listdir(sourceFolder) if f.endswith('_map.txt')]
for f in allFiles:
    with open(f, 'r') as fp:
        for line in fp:
            line = line.strip()
            if not line:
              continue
            position = line.index(':')
            key = line[0:position]
            value = int(line[position +1:])
            result[key] = result.get(key,0) + value
with open(targetFile, 'w') as fp:
    for k,v in result.items():
        fp.write(k + ':' + str(v) + '\n')

if __name__ == '__main__':
    Reduce('test', 'test\\result.txt')
```

保存并运行上述程序，首先运行 FileSplit. py，将文件切分，生成若干小文件，如图 11-2 所示。



图 11-2　文件切分结果

然后运行 Map. py 程序，得到中间结果，如图 11-3 所示。

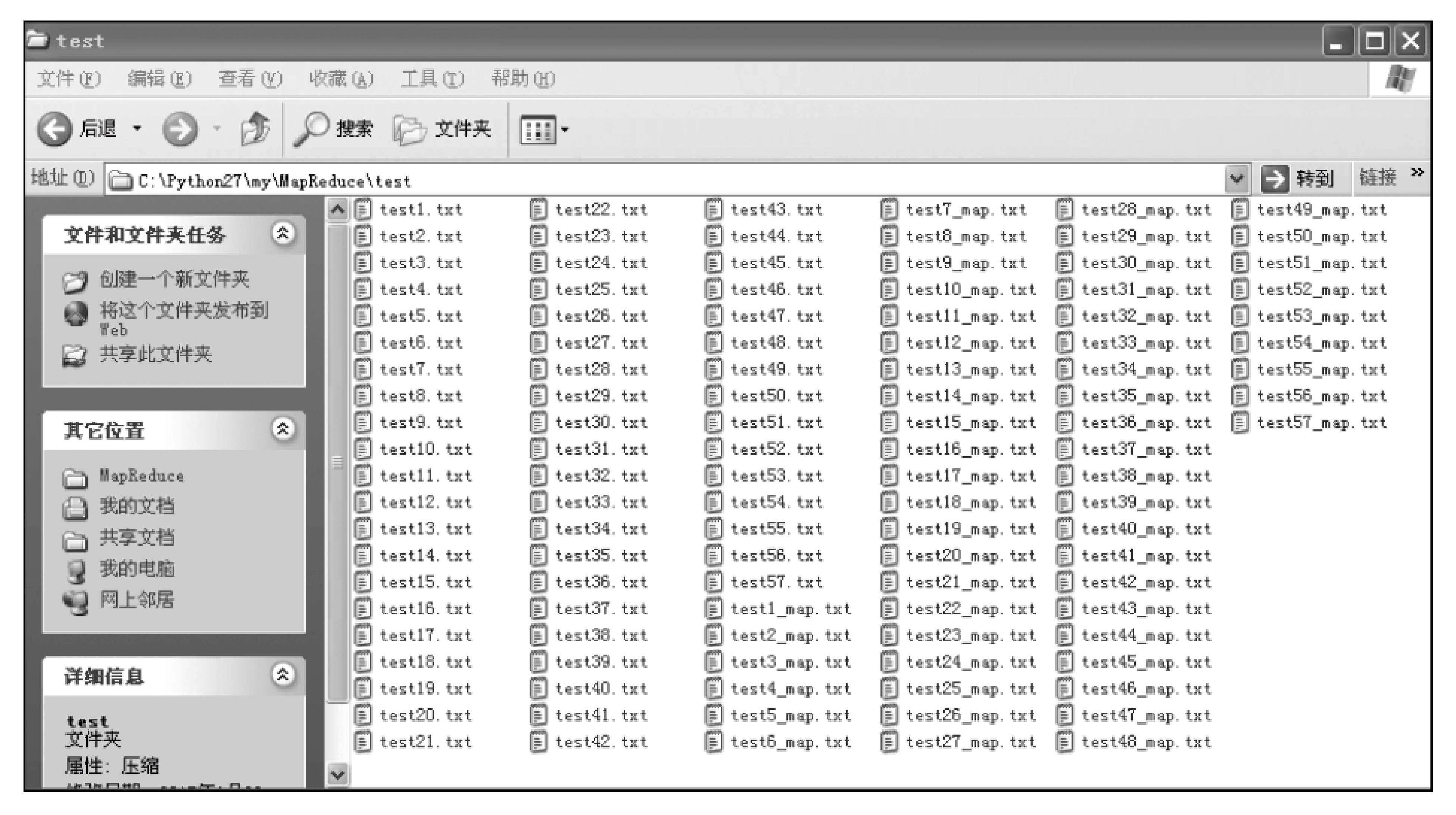


图 11-3　运行 Map. py 程序后的结果

最后运行 Reduce. py 程序，得到最终结果，如图 11-4 所示。



图 11-4　运行 Reduce. py 程序后的结果

本章知识精要

（1）Amazon、Google 等很多大公司都推出了自己的大数据框架和服务，比较流行的有 MapReduce、Hadoop 和 Spark。

（2）MapReduce 编程的思路：首先对大数据进行分割，切分为一定大小的小数据；然后将分割的数据交给多个 Mapper 函数进行处理，Mapper 函数处理后将产生一组规模较小的数据，多个规模较小的数据再提交给 Reducer 函数进行处理，得到一个更小规模的数据或者最终结果。

习　　题

1. 简单介绍常见的大数据处理框架。
2. 运行本章中代码并理解 MapReduce 编程思路。

第 12 章　Windows 系统编程

Python 是一门强大的脚本语言，它可以把其他语言编写的程序黏合在一起，可以很容易地调用外部程序，以及调用其他语言编写的动态链接库中的代码，甚至还可以将 Python 程序打包为.exe 可执行程序以便在没有安装 Python 的 Windows 系统中运行。在本章中通过大量示例来介绍 Windows 平台的混合编程技术以及底层编程技术，不过有些内容可能需要读者对 Windows 平台有较深层的了解，您可以查阅《Windows 内核原理与实现》、《Windows 核心编程》、《深入解析 Windows 操作系统》或其他相关书籍。

在本章中大部分程序主要使用到 pywin32、py2exe、ctypes 和其他几个扩展库。由于篇幅问题，主要通过一些示例来演示 Python 进行 Windows 底层编程以及混合编程的思路，并没有深入展开介绍相关的理论知识，而是假设读者已经了解或者可以通过查阅相关资料进行学习。

12.1　注册表编程

对于 Windows 操作系统，注册表无疑是非常重要的组成部分，Windows 将几乎所有软、硬件系统配置信息都保存在注册表中。通过读取注册表中的数据，可以获取 Windows 平台的相应信息，比如，已安装的服务和程序列表、开机自动运行的程序列表、文件类型与程序的关联关系等；通过修改注册表中的数据，可以对 Windows 系统进行详细的配置。

Windows 注册表有如下 5 个根键。

(1) HKEY_LOCAL_MACHINE (HKLM)。

(2) HKEY_CURRENT_CONFIG (HKCC)。

(3) HKEY_CLASSES_ROOT (HKCR)。

(4) HKEY_USERS (HKU)。

(5) HKEY_CURRENT_USER (HKCU)。

单击“开始”→“运行”命令，弹出“运行”对话框，在对话框中输 regedit.exe 并按 Enter 键，可以打开“注册表编辑器”窗口，如图 12-1 所示，在注册表编辑器界面中可以对注册表的键和值进行增、删、改、查等操作。

在注册表中，值可以为数值、字符串等多种类型，详细类型如表 12-1 所示。

表 12-1　注册表中值的类型

类　型　名	说　　明
REG_NONE	没有类型
REG_SZ	字符串类型
REG_EXPAND_SZ	一个可扩展的字符串值，其中可以包含环境变量

续表

类　型　名	说　　明
REG_BINARY	二进制类型
REG_DWORD / REG_DWORD_LITTLE_ENDIAN	DWORD 类型，用于存储 32 位无符号整数，即 0～4 294 967 295 之间的整数，以 little-endian 格式存储
REG_DWORD_BIG_ENDIAN	DWORD 类型，用于存储 32 位无符号整数，即 0～4 294 967 295 之间的整数，以 big-endian 格式存储
REG_LINK	到其他注册表键的链接，指定根键或到目标键的路径
REG_MULTI_SZ	一个多字符串值，指定一个非空字符串的排序列表
REG_RESOURCE_LIST	资源列表，用于枚举即插即用硬件及其配置
REG_FULL_RESOURCE_DESCRIPTOR	资源标识符，用于枚举即插即用硬件及其配置
REG_RESOURCE_REQUIREMENTS_LIST	资源需求列表，用于枚举即插即用硬件及其配置
REG_QWORD / REG_QWORD_LITTLE_ENDIAN	QWORD 类型，用于存储 64 位无符号整数，以 little-endian 格式存储或未指定存储格式

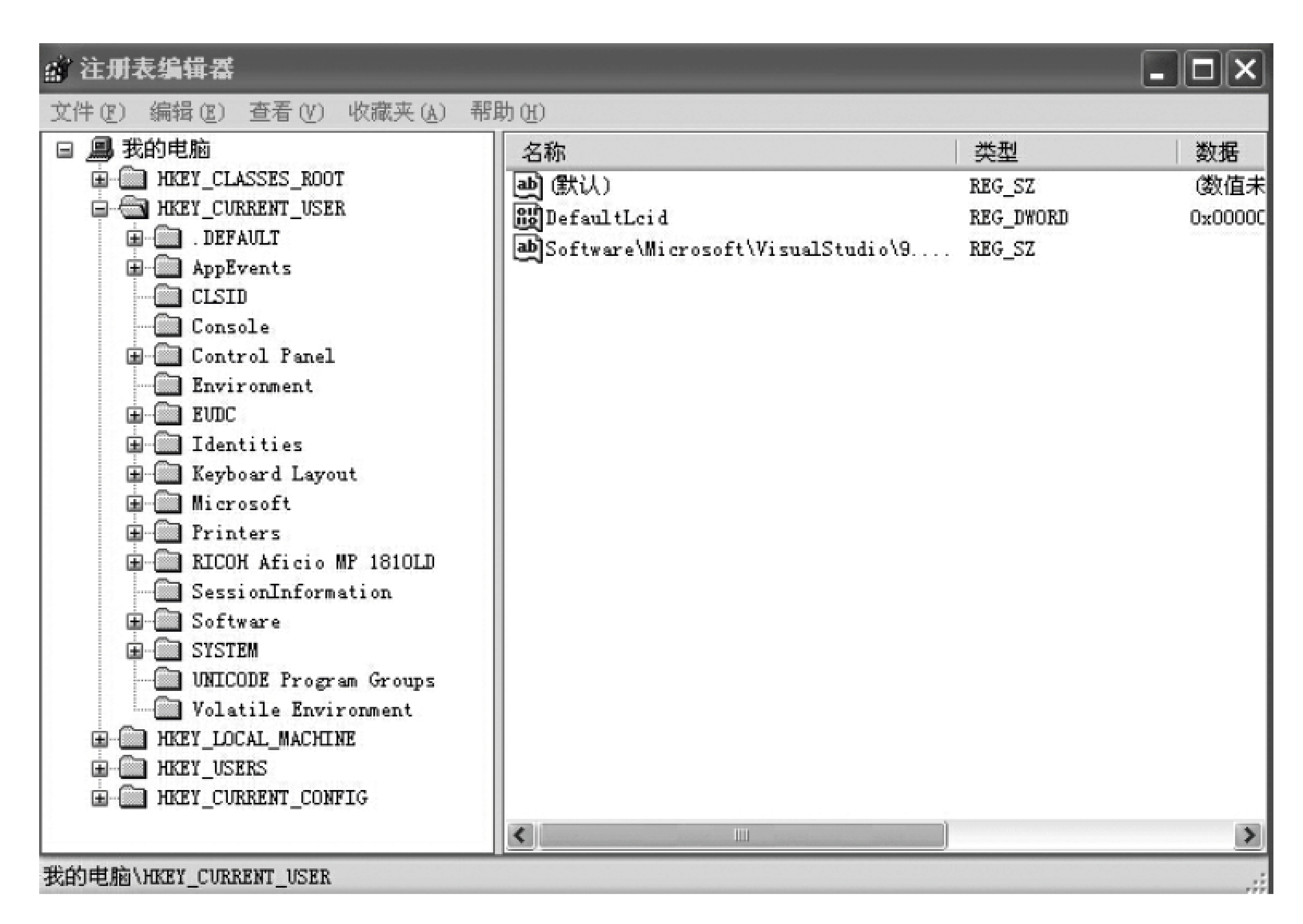


图 12-1　注册表编辑器

对于注册表编程，可以使用 win32api 模块和 win32con 模块，其中 win32api 模块封装了 Windows API 函数，提供了非常友好的接口。该模块中常用的注册表操作函数有 8 个。

(1) RegOpenKey()/RegOpenKeyEx()：打开注册表。

(2) RegCloseKey()：关闭注册表。

(3) RegQueryValue()/RegQueryValueEx()：读取项值。

(4) RegSetValue()/RegSetValueEx()：设置项值。

(5) RegCreateKey()/RegCreateKeyEx():添加项。
(6) RegDeleteKey():删除项。
(7) RegEnumKey():枚举子键。
(8) RegDeletetValue():删除值。
例如,下面的代码用来查询注册表并输出本机安装的 IE 浏览器软件版本信息:

```
>>>import win32api
>>>import win32con
>>> key= win32api.RegOpenKey (win32con.HKEY_LOCAL_MACHINE, 'SOFTWARE\\Microsoft\\Internet Explorer',0,win32con.KEY_ALL_ACCESS)
>>>win32api.RegQueryValue(key,'')
''
>>>win32api.RegQueryValueEx(key,'Version')
('8.0.6001.18702', 1)
>>>win32api.RegQueryInfoKey(key)
(64, 12, 130578396029843750L)
>>>win32api.RegCloseKey(key)
```

下面的代码用来检查随系统启动而启动的程序列表:

```
from win32api import *
from win32con import *

def GetValues(fullname):
    name=str.split(fullname,'\\',1)
    try:
        if name[0]=='HKEY_LOCAL_MACHINE':
            key=RegOpenKey(HKEY_LOCAL_MACHINE,name[1],0,KEY_READ)
        elif name[0]=='HKEY_CURRENT_USER':
            key=RegOpenKey(HKEY_CURRENT_USER,name[1],0,KEY_READ)
        elif name[0]=='HKEY_CURRENT_ROOT':
            key=RegOpenKey(HKEY_CURRENT_ROOT,name[1],0,KEY_READ)
        elif name[0]=='HKEY_CURRENT_CONFIG':
            key=RegOpenKey(HKEY_CURRENT_CONFIG,name[1],0,KEY_READ)
        elif name[0]=='HKEY_USERS':
            key=RegOpenKey(HKEY_USERS,name[1],0,KEY_READ)
        else:
            print 'Error, no key named ',name[0]
        info =RegQueryInfoKey(key)
        for i in range(0,info[1]):
            ValueName =RegEnumValue(key,i)
            print str.ljust(ValueName[0],20),ValueName[1]
        RegCloseKey(key)
    except BaseException,e:
        print 'Sth is wrong'
        print e
```

```
if __name__=='__main__':
    KeyNames= ['HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
              CurrentVersion\\Run',
              'HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
              CurrentVersion\\RunOnce',
              HKEY_LOCAL_MACHINE\\SOFTWARE\\Microsoft\\Windows\\
              CurrentVersion\\RunOnceEx',
              'HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\
              CurrentVersion\\Run',
              'HKEY_CURRENT_USER\\Software\\Microsoft\\Windows\\
              CurrentVersion\\RunOnce']
    for KeyName in KeyNames:
        print KeyName
        GetValues(KeyName)
```

操作 Windows 注册表的另外一种常用方式是使用 Python 模块_winreg,该模块提供了 OpenKey()、DeleteKey()、DeleteValue()、CreateKey()、SetValue()、QueryValueEx()、EnumValue()和 EnumKey()等大量用于注册表访问和操作的方法。下面的代码演示了使用模块_winreg 枚举注册表值的用法:

```
import _winreg
key = _winreg.OpenKey(_winreg.HKEY_CURRENT_USER, r"Software\Microsoft\Windows\
CurrentVersion\Explorer")
try:
    i =0
    while 1:
        Name, Value, Type =_winreg.EnumValue(key, i)
        print repr(Name),':',repr(Value),':',Type
        i +=1
except WindowsError:
    pass
print '='* 20
Name ="FaultTime"
Value, Type =_winreg.QueryValueEx(key, Name)
print Name,Value
```

12.2　创建可执行文件

将 Python 程序转换为.exe 版本可执行程序之后再发布,可以在没有安装 Python 环境的 Windows 平台上运行,这个功能极大地方便了用户。为了将 Python 程序转换为.exe 可执行文件,需要用到 py2exe 和 distutils 模块。当然,首先应保证您编写的 Python 程序可以正常运行,并且本机已安装了所有需要的扩展模块和相关的动态链接库文件。

例如,将 12.1 节最后的代码保存为文件 CheckAndViewAutoRunsInSystem.py,然后编写 setup.py 文件,内容为

```
import distutils
import py2exe
distutils.core.setup(console= ['CheckAndViewAutoRunsInSystem.py'])
```

最后在命令提示符下执行如下命令：

```
python setup.py py2exe
```

接下来就会看到控制台窗口中大量的提示内容飞快地闪过，这个过程将自动搜集 CheckAndViewAutoRunsInSystem.py 程序执行所需要的所有支持文件，如果创建成功的话则会在当前文件夹下生成一个 dist 子文件夹，其中包含了最终程序执行所需要的所有内容。等待编译完成以后，将 dist 文件中的文件打包发布即可。例如，上面步骤完成之后，dist 文件夹中的文件列表如图 12-2 所示。

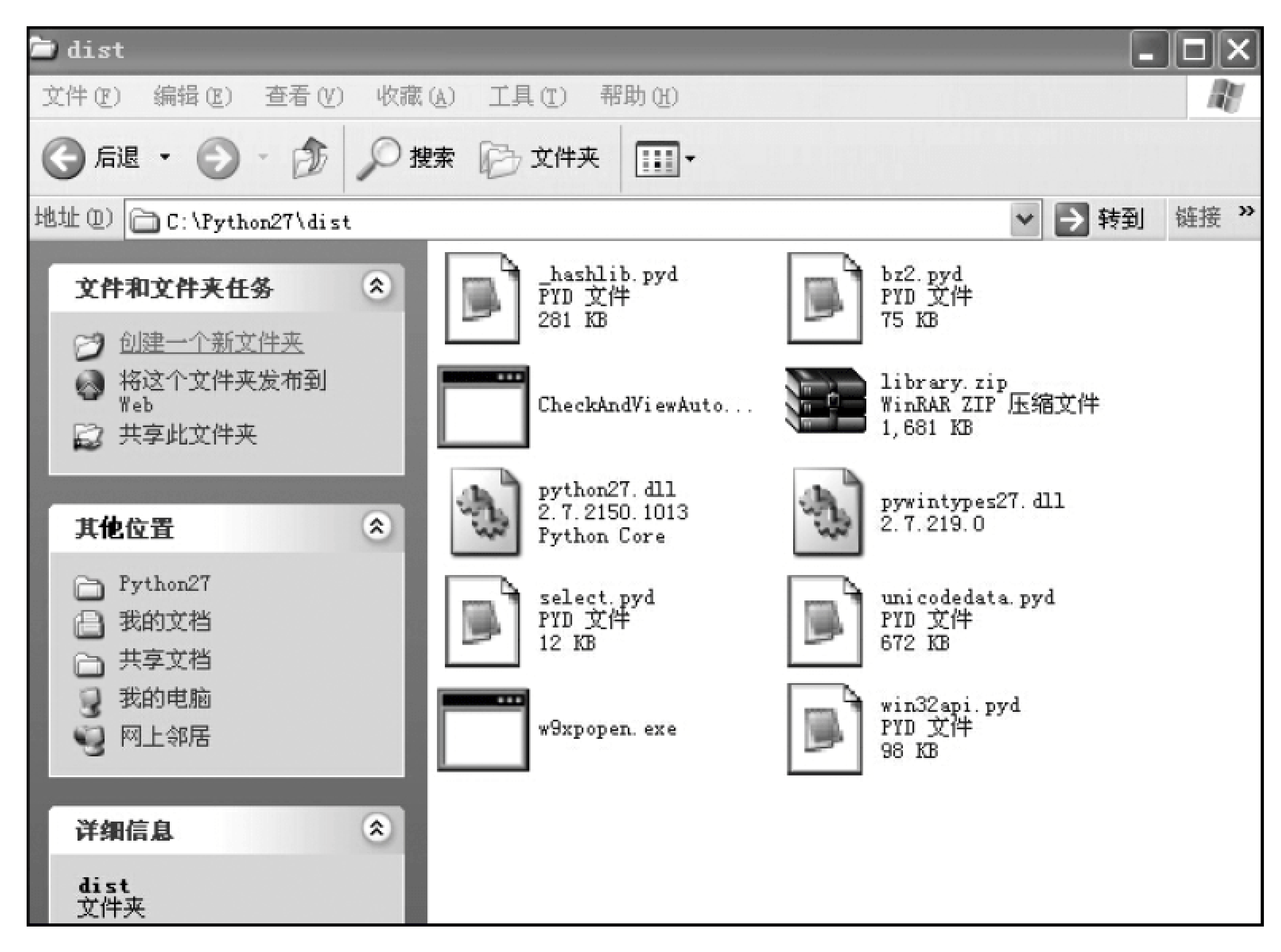


图 12-2　dist 文件夹中的文件列表

py2exe 模块的详细用法可以查阅有关资料，但是对于一般应用而言，上面的代码已经足够了。唯一要注意的问题是，对于控制台应用程序，要想转换为.exe 可执行程序直接套用上面的代码框架即可，仅需要把

```
distutils.core.setup(console= ['CheckAndViewAutoRunsInSystem.py'])
```

这行代码中的文件名替换为自己的 Python 程序文件名即可。对于 GUI 应用程序，则应该将上面代码中的关键字 console 修改为 windows。

12.3　调用外部程序

在 Python 中可以通过多种不同的方法来调用其他语言编写的外部程序或系统内置命令以及动态链接库中的代码，下面简单介绍其中的 4 种。

1. 使用 os 模块的方法调用外部程序

```
>>> import os
>>> os.system('notepad.exe')
0
>>> os.system('notepad C:\\dir.txt')
0
```

使用上面的 system()方法也可以调用 Windows 系统命令，如 dir、xcopy 等，但是有一个缺点，不论启动什么程序会先启动一个黑色控制台窗口，然后再打开被调程序，如图 12-3 所示。

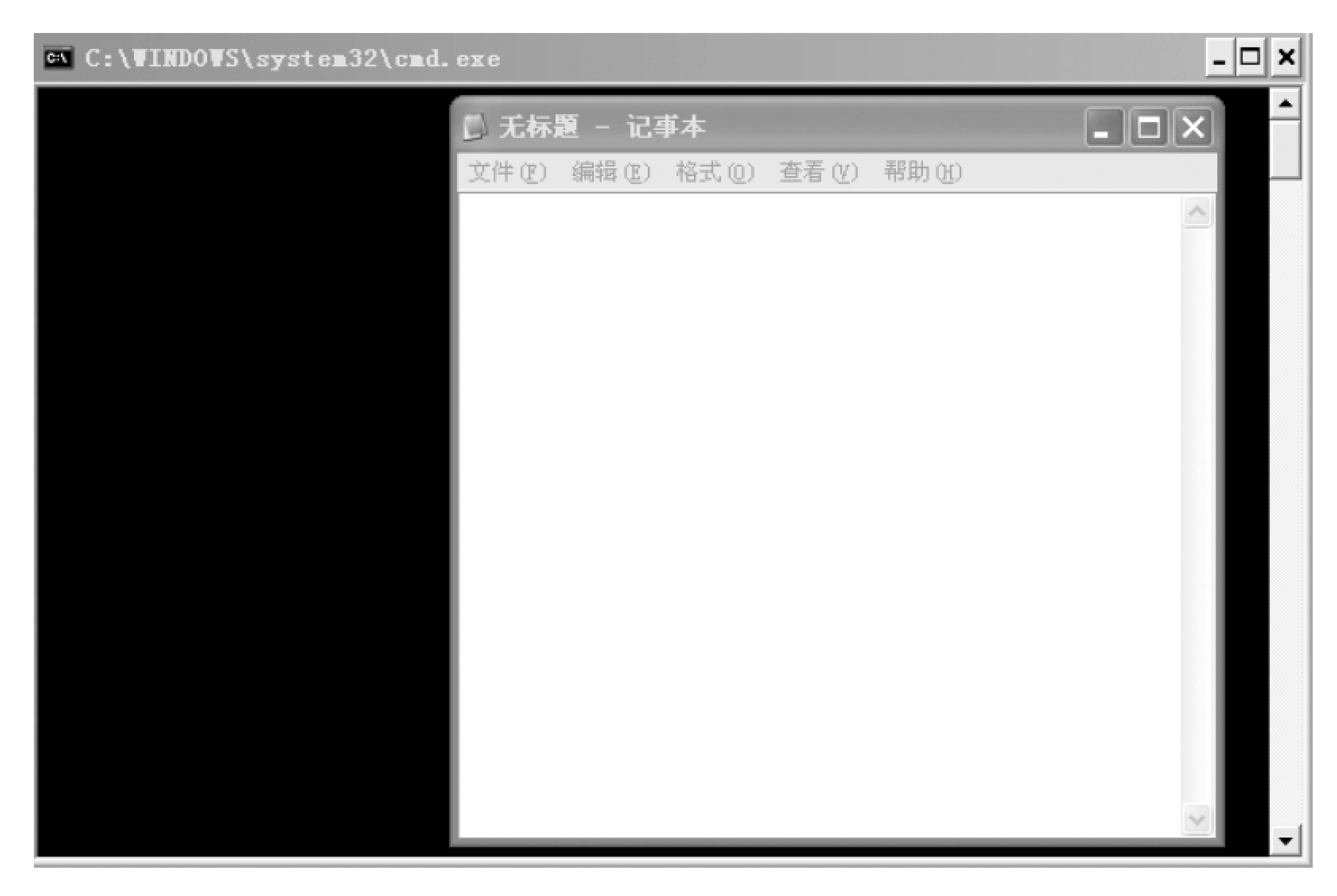


图 12-3 使用 os 模块的 system()方法启动记事本

也可以使用 os 模块的 popen()方法来打开外部程序，这样不会出现黑色的命令提示符窗口。

```
>>> os.popen(r'C:\windows\notepad.exe')
<open file 'C:\\windows\\notepad.exe', mode 'r' at 0x012BEF98>
```

或者，还可以使用 os 模块的 startfile()方法来打开外部程序或文件，系统将自动关联相应的程序来打开或执行文件。

```
>>> import os
>>> os.startfile(r'C:\windows\notepad.exe')
>>> os.startfile(r'wxIsPrime.py')
```

2. 使用 win32api 模块调用 ShellExecute()函数来启动外部程序

```
>>> import win32api
>>> win32api.ShellExecute(0, 'open', 'notepad.exe', '', '',0)          #0 表示后台运行程序
42
```

```
>>>win32api.ShellExecute(0, 'open', 'notepad.exe', '', '',1)          #1表示前台运行程序
42
>>>win32api.ShellExecute(0,'open','notepad.exe','C:\\dir.txt', '',1)
                                                                    #传递参数打开指定文件
42
>>>win32api.ShellExecute(0, 'open', 'www.python.org','','',1)          #打开网址
42
>>>win32api.ShellExecute(0, 'open',r'C:\dir.txt', '', '',1)           #相当于双击文件
42
```

使用这种方式运行程序或者打开文件时，不会像 os 模块的 system()方法那样先打开一个命令提示符窗口，并且系统将根据文件类型自动关联相应程序并打开文件，类似于在资源管理器中双击打开文件或单击打开超链接。例如，如果打开的是记事本文件，则会自动使用记事本程序打开；如果指定的是个域名，则会自动使用默认浏览器打开该网址；如果打开的是可执行文件，则会自动打开并运行该程序文件。

3. 通过创建进程来启动外部程序

```
>>>import win32process
>>> handle = win32process.CreateProcess (r'C:\windows\notepad.exe','', None, None, 0,
win32process.CREATE_NO_WINDOW, None, None,win32process.STARTUPINFO())
                                                                    #打开记事本程序
>>>win32process.TerminateProcess(handle[0],0)                       #关闭刚才打开的程序
>>> handle = win32process.CreateProcess (r'C:\windows\notepad.exe', '', None, None, 0,
win32process.CREATE_NO_WINDOW,None,None,win32process.STARTUPINFO())
>>>import win32event
>>>win32event.WaitForSingleObject(handle[0],-1)                     #需要手动关闭记事本
0
```

4. 通过 ctypes 来调用动态链接库代码

ctypes 是 Python 处理动态链接库的标准扩展模块，提供了与 C 语言兼容的数据类型，允许在 Python 程序中调用动态链接库或共享库中的代码，从而支持 Python 与其他编程语言的混合编程，充分发挥各自的优势，大幅度提高开发效率和运行效率。另外，NumPy 模块也提供了一个函数 numpy.ctypeslib.load_library()用来打开指定的动态链接库并返回一个 ctypes 对象，通过该对象可以访问动态链接库中的函数。或者，使用 SciPy 库的 Weave 模块也可以方便地将 C++ 程序以字符串的形式嵌入到 Python 程序中。由于篇幅问题，关于混合编程更多的资料，读者可以查阅相关资料进行学习，本章重点介绍如何使用 ctypes 扩展模块调用动态链接库中的代码。

ctypes 提供了 3 种方法调用动态链接库：cdll、windll 和 oledll，它们的不同之处在于函数调用时的参数传递方式和返回时栈的平衡方式。cdll 加载的库导出的函数必须使用标准的 cdecl 调用约定(函数的参数从右往左依次压入栈内，在函数执行完成后，由函数的调用者负责函数的栈帧平衡)，windll 方法加载的库导出的函数必须使用 stdcall 调用约定(Win32 API 的原生约定)，oledll 方法和 windll 类似，不过假设函数返回一个 HRESULT

错误代码。

下面的代码调用 Windows 动态链接库 user32.dll 中的 MessageBoxA()函数来显示对话框：

```
>>> import ctypes                            #通过 ctypes 可以调用动态链接库中的函数
>>> user32=ctypes.windll.LoadLibrary('user32.dll')
>>> user32.MessageBoxA(0,str.encode('Hello world!'),str.encode('Python
ctypes'),0)
1
```

或者使用下面更为简洁的形式：

```
>>> import ctypes
>>> ctypes.windll.user32.MessageBoxA(0,str.encode('Hello world!'),str.encode('Python
ctypes'),0)
```

下面的代码调用标准 C 函数库 msvcrt 中的 printf()函数来输出文本：

```
import ctypes
msvcrt=ctypes.cdll.LoadLibrary('msvcrt')
printf=msvcrt.printf
printf('Hello world!')
```

或者使用下面形式：

```
import ctypes
ctypes.cdll.msvcrt.printf('Hello world!')
```

该程序需要在命令提示符环境中而不是在 IDLE 中执行，如果在 IDLE 环境中运行输出的是字符数量而不是字符，例如，将上面的代码复制到 IDLE 交互窗口执行结果如下：

```
>>> import ctypes
>>> ctypes.cdll.msvcrt.printf('Hello world!')
12
```

假设将上面的代码保存为 useprintfthroughctypes.py 文件，然后在命令提示符中运行结果如图 12-4 所示。

```
C:\Python27>python useprintfthroughctypes.py
Hello world!
C:\Python27>useprintfthroughctypes.py
Hello world!
C:\Python27>
```

图 12-4　使用 ctypes 库调用 C 语言的 printf 函数

ctypes 提供了与 C 语言兼容的数据类型，但在 Python 中使用 C 语言的结构体时，需要用类来改写。表 12-2 给出了基本类型的对应关系，关于结构体改写的内容可以通过后面给出的示例代码了解大概思路。

表 12-2　基本类型对应关系

ctypes type	C type	Python type
c_bool	_Bool	bool (1)
c_char	char	1-character string
c_wchar	wchar_t	1-character unicode string
c_byte	char	int/long
c_ubyte	unsigned char	int/long
c_short	short	int/long
c_ushort	unsigned short	int/long
c_int	int	int/long
c_uint	unsigned int	int/long
c_long	long	int/long
c_ulong	unsigned long	int/long
c_longlong	__int64 or long long	int/long
c_ulonglong	unsigned __int64 or unsigned long long	int/long
c_float	float	float
c_double	double	float
c_longdouble	long double	float
c_char_p	char * (NUL terminated)	string or None
c_wchar_p	wchar_t * (NUL terminated)	unicode or None
c_void_p	void *	int/long or None

ctypes 是 Python 进行操作系统底层开发的重要技术，借助于该扩展库，可以访问操作系统所有底层功能，例如，下面的代码可以枚举系统当前运行的进程列表：

```
#EnumProcess.py
from ctypes.wintypes import *
from ctypes import *
import collections

kernel32 =windll.kernel32

class tagPROCESSENTRY32(Structure):     #定义结构体
    _fields_ =[('dwSize',              DWORD),
               ('cntUsage',            DWORD),
               ('th32ProcessID',       DWORD),
               ('th32DefaultHeapID',   POINTER(ULONG)),
               ('th32ModuleID',        DWORD),
               ('cntThreads',          DWORD),
```

```
                ('th32ParentProcessID',    DWORD),
                ('pcPriClassBase',         LONG),
                ('dwFlags',                DWORD),
                ('szExeFile',              c_char * 260)]

def enumProcess():
    hSnapshot = kernel32.CreateToolhelp32Snapshot(15, 0)
    fProcessEntry32 = tagPROCESSENTRY32()
    processClass = collections.namedtuple("processInfo","processName processID")
    processSet = []
    if hSnapshot:
        fProcessEntry32.dwSize = sizeof(fProcessEntry32)
        listloop = kernel32.Process32First(hSnapshot, byref(fProcessEntry32))
        while listloop:
            processName = (fProcessEntry32.szExeFile)
            processID = fProcessEntry32.th32ProcessID
            processSet.append(processClass(processName, processID))
            listloop = kernel32.Process32Next(hSnapshot, byref(fProcessEntry32))
    return processSet
for i in enumProcess():
    print(i.processName,i.processID)
```

12.4 创建窗口

在本节中，主要介绍在 Python 中使用 Windows API 函数和 MFC 创建窗口的思路和步骤。

（1）可以调用 Windows 底层 API 函数来创建窗口并构建消息循环，代码如下：

```
import win32gui
from win32con import *
def WndProc(hwnd,msg,wParam,lParam):
    if msg==WM_PAINT:
        hdc,ps = win32gui.BeginPaint(hwnd)
        rect=win32gui.GetClientRect(hwnd)
        win32gui.DrawText(hdc,'GUI Python',len('GUI Python'),rect,
                          DT_SINGLELINE|DT_CENTER|DT_VCENTER)
        win32gui.EndPaint(hwnd,ps)
    if msg == WM_DESTROY:
        win32gui.PostQuitMessage(0)
    return win32gui.DefWindowProc(hwnd, msg, wParam, lParam)

wc = win32gui.WNDCLASS()
wc.hbrBackground = COLOR_BTNFACE + 1
wc.hCursor = win32gui.LoadCursor(0, IDC_ARROW)
```

```
wc.hIcon = win32gui.LoadIcon(0, IDI_APPLICATION)
wc.lpszClassName = 'Python on Windows'
wc.lpfnWndProc = WndProc

reg = win32gui.RegisterClass(wc)

hwnd = win32gui.CreateWindow(
    reg, 'Python', WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, CW_USEDEFAULT,
    CW_USEDEFAULT, CW_USEDEFAULT, 0, 0, 0, None)
win32gui.ShowWindow(hwnd, SW_SHOWNORMAL)
win32gui.UpdateWindow(hwnd)

win32gui.PumpMessages()
```

（2）使用 MFC 创建窗口，并创建菜单。

```
import win32ui
import win32api
from win32con import *
from pywin.mfc import window

class MyWnd(window.Wnd):
    def __init__(self):
        window.Wnd.__init__(self, win32ui.CreateWnd())
        self._obj_.CreateWindowEx(WS_EX_CLIENTEDGE,
                                  win32ui.RegisterWndClass(0, 0, COLOR_WINDOW+1),
                                  'MFC GUI', WS_OVERLAPPEDWINDOW,
                                  (10,10,800,500), None, 0, None)

        self.HookMessage(self.OnRClick, WM_RBUTTONDOWN)

        submenu = win32ui.CreateMenu()
        menu = win32ui.CreateMenu()
        submenu.AppendMenu(MF_STRING, 1051, '&Open')
        submenu.AppendMenu(MF_STRING, 1052, '&Close')
        submenu.AppendMenu(MF_STRING, 1053, '&Save')
        menu.AppendMenu(MF_STRING | MF_POPUP, submenu.GetHandle(), '&File')

        submenu = win32ui.CreateMenu()
        submenu.AppendMenu(MF_STRING, 1054, '&Copy')
        submenu.AppendMenu(MF_STRING, 1055, '&Paste')
        submenu.AppendMenu(MF_SEPARATOR, 1056, None)
        submenu.AppendMenu(MF_STRING, 1057, 'C&ut')
        menu.AppendMenu(MF_STRING | MF_POPUP, submenu.GetHandle(), '&Edit')
```

```
        submenu = win32ui.CreateMenu()
        submenu.AppendMenu(MF_STRING, 1058, 'Tools')
        submenu.AppendMenu(MF_STRING | MF_GRAYED, 1059, 'Settings')
        m = win32ui.CreateMenu()
        m.AppendMenu(MF_STRING | MF_POPUP | MF_CHECKED, submenu.GetHandle(), 'Option')
        menu.AppendMenu(MF_STRING | MF_POPUP, m.GetHandle(), '&Other')

        self._obj_.SetMenu(menu)
        self.HookCommand(self.MenuClick, 1051)
        self.HookCommand(self.MenuClick, 1052)
        self.HookCommand(self.MenuClick, 1053)
        self.HookCommand(self.MenuClick, 1054)
        self.HookCommand(self.MenuClick, 1060)

    def OnRClick(self,param):
        submenu = win32ui.CreatePopupMenu()
        submenu.AppendMenu(MF_STRING, 1060, 'Copy')
        submenu.AppendMenu(MF_STRING, 1061, 'Paste')
        submenu.AppendMenu(MF_SEPARATOR, 1062, None)
        submenu.AppendMenu(MF_STRING, 1063, 'Cut')
        submenu.TrackPopupMenu(param[5], TPM_LEFTALIGN | TPM_LEFTBUTTON | TPM_RIGHTBUTTON,
        self)

    def MenuClick(self, lParam, wParam):
        if lParam == 1051:
            self.MessageBox('Open', 'Python', MB_OK)
        elif lParam == 1053:
            self.MessageBox('Save', 'Python', MB_OK)
        elif lParam == 1052:
            self.OnClose()
        elif lParam == 1060 or lParam == 1054:
            self.MessageBox('Copy', 'Python', MB_OK)

    def OnClose(self):
        self.EndModalLoop(0)

    def OnPaint(self):
        dc, ps = self.BeginPaint()
        dc.DrawText('MFC GUI', self.GetClientRect(),
                    DT_SINGLELINE | DT_CENTER | DT_VCENTER)
        self.EndPaint(ps)

w = MyWnd()
w.ShowWindow()
```

```
w.UpdateWindow()
w.RunModalLoop(1)
```

下面的代码演示了创建 MFC 窗口并响应按钮消息的过程：

```
import win32ui
import win32con
from pywin.mfc import dialog

class MyDialog(dialog.Dialog):
    def OnInitDialog(self):
        dialog.Dialog.OnInitDialog(self)
        self.HookCommand(self.OnButton1,1051)
        self.HookCommand(self.OnButton2,1052)
    def OnButton1(self, wPrarm, lParam):
        win32ui.MessageBox('Button1', 'Python', win32con.MB_OK)
        # self.EndDialog(1)
    def OnButton2(self, wParam, lParam):
        text = self.GetDlgItemText(1054)
        win32ui.MessageBox(text, 'Python', win32con.MB_OK)
        # self.EndDialog(1)

style = win32con.DS_MODALFRAME | win32con.WS_POPUP | win32con.WS_VISIBLE \
    | win32con.WS_CAPTION | win32con.WS_SYSMENU | win32con.DS_SETFONT
childstyle = win32con.WS_CHILD | win32con.WS_VISIBLE
buttonstyle = win32con.WS_TABSTOP | childstyle

di = ['Python', (0,0,300,180), style, None, (8,'MS Sans Serif')]
Button1 = (['Button', 'Button1', 1051, (80, 150, 50, 14),
          buttonstyle | win32con.BS_PUSHBUTTON])
Button2 = (['Button', 'Button2', 1052, (160, 150, 50, 14),
          buttonstyle | win32con.BS_PUSHBUTTON])
Stadic = (['Static', 'Python Dialog', 1053, (130, 50, 60, 14), childstyle])
Edit = (['Edit', '', 1054, (130, 80, 60, 14),
    childstyle | win32con.ES_LEFT | win32con.WS_BORDER | win32con.WS_TABSTOP])

init = []
init.append(di)
init.append(Button1)
init.append(Button2)
init.append(Stadic)
init.append(Edit)

mydialog = MyDialog(init)
mydialog.DoModal()
```

12.5 判断操作系统的版本

某些情况下，程序可能依赖于特定版本操作系统中的功能或者希望程序在不同版本的操作系统中有不同的表现，因此能够在程序运行时获知操作系统的版本就变得非常有必要。Python 支持使用多种不同的方法来获取操作系统的版本信息。

下面的代码演示了使用不同的方法来获取本机 Windows 操作系统的版本信息的用法：

```
>>> import os
>>> print(os.popen('ver').read())
Microsoft Windows [版本 6.1.7601]
>>> import sys
>>> print(sys.getwindowsversion())
sys.getwindowsversion(major=6, minor=1, build=7601, platform=2, service_pack='Service
Pack 1')
>>> import platform
>>> print(platform.platform())
Windows-7-6.1.7601-SP1
```

Windows 管理规范(Windows Management Instrumentation，WMI)是 Windows 的一项核心技术，它以公共信息模型对象管理器(Common Information Model Object Manager，CIMOM)为基础，是一个描述 Windows 操作系统构成单元的对象数据库。WMI 是 Windows 的核心组件，通过编写 WMI 脚本和应用程序可以获取计算机系统、软件和硬件信息，还可以对计算机进行管理，比如关机、重新启动计算机等。

```
>>> import wmi
>>> wmiShell = wmi.WMI()
>>> print(wmiShell.Win32_OperatingSystem()[0].Caption)
Microsoft Windows 7 旗舰版
```

还可以通过 os.system('ver')语句来查看 Windows 操作系统的版本，但需要编写程序并在命令提示符环境中运行，在 IDLE 环境中运行无法查看结果。

本章知识精要

(1) pywin32 封装了 Windows 底层的几乎所有 API 函数。

(2) 使用 ctypes 模块可以调用任意其他语言编写的动态链接库或共享库文件。

(3) 使用 py2exe 可以方便地将 Python 程序转换为.exe 程序。

(4) 可以通过 os 模块的 system()、popen()和 startfile()等方法方便地调用外部程序或打开磁盘上的文件，也可以使用 win32api 模块的 ShellExecute()方法或 win32process 模块的 CreateProcess()方法实现这一目的。

(5) ctypes 提供了 3 种调用动态链接库文件的方法，分别为 cdll、windll 和 oledll，它们的不同之处在于调用函数时的参数传递方式和函数返回时的栈平衡方式。

(6) 在 Windows 平台上,可以通过 sys 模块的 getwindowsversion()、platform 模块的 platform()以及 wmi 模块等多种方法来动态检测系统版本。

习　题

1. 查阅相关资料,解释注册表几大根键的用途。
2. 选择一个编写好的 Python 程序,将其转换为.exe 可执行文件。
3. 编写代码,使用至少 3 种不同的方法启动 Windows 自带的计算器程序。
4. 编写代码,检测您所使用的操作系统版本。

第 13 章　多线程编程

由于硬件技术的飞速发展，早期的多核、多处理器等高端技术已经走进了普通家庭，再加上内存、主频、硬盘等各种硬件配置的飞速提高，大幅度提高了普通 PC 的运算速度和数据处理能力。在多核、多处理器平台上，每个核可以运行一个线程，多个线程同时运行并相互协作，从而达到高速处理任务的目的。

然而，即使是高端服务器或工作站甚至集群系统，处理器和核的数量总是有限的，如果线程的数量多于核的数量，就必然需要进行调度。在调度时，处理器为每个线程分配一个很短的时间片，所有线程根据具体的调度算法轮流获得该时间片。当时间片用完以后，即使该线程还没有执行完也要退出处理器并等待下次调度。由于处理器中寄存器的数量有限，而不同的线程很可能需要使用到相同的一组寄存器来保存中间计算结果或当前状态。因此，在调度线程时必须要做好上下文保存和恢复工作，以保证该线程下次被调度进处理器后能够继续上次的工作。虽然这些工作并不需要 Python 程序员操心，但是我们必须清楚的一件事是，并不是使用的线程数量越多越好，如果线程太多的话，线程调度带来的开销可能会比线程实际执行的开销还大，这样使用多线程就失去本来的意义了。

13.1　threading 模块

threading 模块是 Python 支持多线程编程的重要模块，该模块是在底层模块_thread 的基础上开发的更高层次的线程编程接口，提供了大量的方法和类来支持多线程编程，极大地方便了用户。threading 模块常用方法如表 13-1 所示。

表 13-1　threading 模块常用方法

方　　法	功能说明
threading. active_count()	返回当前处于 alive 状态的 Thread 对象数量
threading. current_thread()	返回当前 Thread 对象
threading. get_ident()	返回当前线程的线程标识符。线程标识符是一个非负整数，并没特殊含义，只是用来标识线程，该整数可能会被循环利用。Python 3.3 及以后版本支持该方法
threading. enumerate()	返回当前处于 alive 状态的所有 Thread 对象列表
threading. main_thread()	返回主线程对象，即启动 Python 解释器的线程对象。Python 3.4 及以后版本支持该方法
threading. stack_size([size])	返回创建线程时使用的栈的大小，如果指定 size 参数，则用来指定后续创建的线程使用的栈大小，size 必须是 0(表示使用系统默认值)或大于 32K 的正整数

下面的代码简单演示了该模块方法的用法：

```
>>> import threading
>>> threading.stack_size()              #查看当前线程栈的大小
0
>>> threading.stack_size(64 * 1024)     #设置当前线程栈的大小
0
>>> threading.stack_size()
65536
>>> threading.active_count()
2
>>> threading.current_thread()
<_MainThread(MainThread, started 4852)>
>>> threading.enumerate()
[<Thread(SockThread, started daemon 9620)>, <_MainThread(MainThread, started 4852)>]
```

13.2 Thread 对象

threading 模块提供了 Thread、Lock、RLock、Condition、Event、Timer 和 Semaphore 等大量类来支持多线程编程，Thread 是其中最重要也是最基本的一个类，可以通过该类创建线程并控制线程的运行。本节首先介绍 Thread 对象，然后在 13.3 节介绍线程同步技术以及 Lock、RLock、Condition 和 Event 等对象的用法。

Thread 类支持使用两种方法来创建线程：一种方法是为构造函数传递一个可调用对象；另一种方法是继承 Thread 类并在派生类中重写__init__()和 run()方法。创建线程对象以后，可以调用其 start()方法来启动，该方法自动调用该类对象的 run()方法，此时该线程处于 alive 状态，直至线程的 run()方法运行结束。Thread 对象成员如表 13-2 所示。

表 13-2 Thread 对象成员

成　　员	说　　明
start()	自动调用 run()方法，启动线程，执行线程代码
run()	线程代码，用来实现线程的功能与业务逻辑，可以在子类中重写该方法来自定义线程的行为
__init__(self, group=None, target=None, name=None, args=(), kwargs=None, verbose=None)	构造函数
name	用来读取或设置线程的名字
ident	线程标识，非 0 数字或 None(线程未被启动)
is_alive()、isAlive()	测试线程是否处于 alive 状态
daemon	布尔值，表示线程是否为守护线程
join(timeout=None)	等待线程结束或超时返回

13.2.1 Thread 对象中的方法

(1) join([timeout])：阻塞当前线程，等待被调线程结束或超时后再继续执行当前线程

的后续代码,参数 timeout 用来指定最长等待时间,单位为秒。

```
import threading
import time
def func1(x, y):
    for i in range(x, y):
        print i,
    time.sleep(10)

t1=threading.Thread(target=func1, args=(15, 20))
t1.start()
t1.join(5)
t2=threading.Thread(target=func1, args=(5, 10))
t2.start()
```

保存并运行上面的程序将会发现,首先输出 15～19 这 5 个整数,然后程序暂停,几秒钟以后又继续输出 5～9 这 5 个整数。如果将 t1.join(5)这一行注释掉再运行,两个线程的输出将会重叠在一起,这是因为两个线程并发运行,而不是第一个结束以后再运行第二个。可以说,这是线程同步的一个最简单的形式。当然,还可以把 time.sleep(10)这一行注释掉再运行,会发现两个线程的输出之间没有时间间隔,您能想明白其中的原因吗?

(2) isAlive():测试线程是否处于运行状态。

```
import threading
import time
def func1(x, y):
    for i in range(x, y):
        print i
    #time.sleep(10)

t1=threading.Thread(target=func1, args=(15, 20))
t1.start()
t1.join(5)          #注释掉这里试试
t2=threading.Thread(target=func1, args=(5, 10))
t2.start()
t2.join()           #注释掉这里试试
print 't1:',t1.isAlive()
print 't2:',t2.isAlive()
```

运行上面的程序会发现,最后两个的输出都是 False,即两个线程都执行完了。如果将 t1.join(5)这一行注释掉会发现最后两行的输出结果没有变化,这是因为 t2.join()这一行代码会阻塞当前程序直至线程 t2 运行结束,对于本例中的线程 t1 基本也运行结束了。如果将线程 t1 的参数范围增大则会发现,倒数第二行的输出结果很可能会变为 True。请您再单独将 t2.join()这一行注释掉而保留 t1.join(5)再运行程序,尝试着理解运行结果,当然,为了验证您的答案,可以将线程的参数范围适当变大和变小。

13.2.2 Thread 对象中的 daemon 属性

在脚本运行过程中有一个主线程，若在主线程中创建了子线程，当主线程结束时根据子线程 daemon 属性值的不同可能会发生下面的两种情况之一：①当某子线程的 daemon 属性为 False 时，主线程结束时会检测该子线程是否结束，如果该子线程尚未完成，则主线程会等待它完成后再退出；②当某子线程的 daemon 属性为 True 时，主线程运行结束时不对该子线程进行检查而直接退出，同时所有 daemon 值为 True 的子线程将随主线程一起结束，而不论是否运行完成。daemon 属性的值默认为 False，如果需要修改，则必须在调用 start()方法启动线程之前进行修改。

以上论述不适用于 IDLE 环境中的交互模式或脚本运行模式，因为在该环境中的主线程只有在退出 Python IDLE 时才终止。

```
import threading
import time

class mythread(threading.Thread):
    def __init__(self, num, threadname):
        threading.Thread.__init__(self, name = threadname)
        self.num = num
        # self.daemon = True
    def run(self):
        time.sleep(self.num)
        print self.num

t1 =mythread(1, 't1')
t2 =mythread(5, 't2')
t2.daemon = True
print t1.daemon
print t2.daemon

t1.start()
t2.start()
```

将上面的代码存储为 ThreadDaemon. py 文件，在 IDLE 环境中运行结果如图 13-1 所示，在命令提示符环境中运行结果如图 13-2 所示。

```
================================ RESTART
>>>
False
True
>>> 1
5
```

图 13-1 ThreadDaemon. py 程序在 IDLE 环境中的运行结果

```
D:\教学课件\Python\code\多线程编程>threaddaemon.py
False
True
1

D:\教学课件\Python\code\多线程编程>
```

图 13-2　ThreadDaemon.py 程序在命令提示符环境中的运行结果

13.3　线程同步技术

毫无疑问，多线程是为了充分利用硬件资源尤其是 CPU 资源来提高任务处理速度和效率的技术。将任务拆分成互相协作的多个线程同时运行，那么属于同一个任务的多个线程之间必然会有交互和同步以便互相协作地完成任务。另外，还可以使用多线程来为用户提供很多的方便。例如，打开软件时可能需要加载大量的模块和库，这可能需要较长的时间，此时可以使用一个线程来显示一个小动画来表示当前软件正在启动，而当后台线程加载完所有的模块和库之后，结束该动画的播放并打开软件主界面，这是多线程同步的一个典型应用。再例如，字处理软件可以使用一个线程来接受用户键盘输入，而使用一个后台线程来进行拼写检查以及字数统计之类的功能并实时将结果显示在状态栏上，这无疑会极大方便用户的使用，对于提高用户体验有重要帮助。类似的多线程应用案例还有很多，您是不是还能再想起来几个呢？

Python 的 threading 模块提供了多种用于线程同步的对象，在本节中将一一进行介绍。

13.3.1　Lock/RLock 对象

Lock 是比较低级的同步原语，当被锁定以后不属于特定的线程。一个锁有两种状态：locked 和 unlocked。如果锁处于 unlocked 状态，acquire()方法将其修改为 locked 并立即返回；如果锁已处于 locked 状态，则阻塞当前线程并等待其他线程释放锁，然后将其修改为 locked 并立即返回。release()方法用来将锁的状态由 locked 修改为 unlocked 并立即返回，如果锁状态本来已经是 unlocked，调用该方法将会抛出异常。

可重入锁 RLock 对象也是一种常用的线程同步原语，可被同一个线程 acquire()多次。当处于 locked 状态时，某线程拥有该锁；当处于 unlocked 状态时，该锁不属于任何线程。RLock 对象的 acquire()/release()调用对可以嵌套，仅当最后一个或者最外层的 release()执行结束后，锁才会被设置为 unlocked 状态。

下面的代码演示了使用 Lock/RLock 对象实现线程同步的思路和步骤：

```
import threading
import time
class mythread(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
    def run(self):
        global x
        lock.acquire()
```

```
            for i in range(3):
                x = x + i
            time.sleep(2)
            print x
            lock.release()
lock = threading.RLock() #lock = threading.Lock()
tl = []
for i in range(10):
    t = mythread()
    tl.append(t)
x = 0
for i in tl:
    i.start()
```

为了节省篇幅，这里就不给出运行结果了，您可以运行上面的程序，然后把 lock. acquire()和 lock. release()这两行注释掉再运行，比较两次结果的不同。

13.3.2 Condition 对象

使用 Condition 对象可以在某些事件触发后才处理数据，可以用于不同线程之间的通信或通知，以实现更高级别的同步。Condition 对象除了具有 acquire()和 release()方法之外，还有 wait()、notify()和 notify_all()等方法。下面通过经典生产者-消费者问题来演示 Condition 对象的用法。

首先实现生产者线程类：

```
import threading

class Producer(threading.Thread):
    def __init__(self, threadname):
        threading.Thread.__init__(self,name=threadname)
    def run(self):
        global x
        con.acquire()
        if x == 20:
            con.wait()
        else:
            print '\nProducer:',
            for i in range(20):
                print x,
                x=x + 1
            print x
            con.notify()
        con.release()
```

接下来实现消费者线程类：

```
class Consumer(threading.Thread):
```

```
    def __init__(self, threadname):
        threading.Thread.__init__(self, name = threadname)
    def run(self):
        global x
        con.acquire()
        if x == 0:
            con.wait()
        else:
            print '\nConsumer:',
            for i in range(20):
                print x,
                x = x - 1
            print x
            con.notify()
        con.release()
```

创建 Condition 对象以及生产者线程和消费者线程。

```
con = threading.Condition()
x = 0
p = Producer('Producer')
c = Consumer('Consumer')
p.start()
c.start()
p.join()
c.join()
print 'After Producer and Consumer all done:',x
```

该程序的运行结果如图 13-3 所示。

```
>>> ================================ RESTART ========================
>>>

Producer: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Consumer: 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

After Producer and Consumer all done: 0
```

图 13-3　使用 Condition 实现线程同步

13.3.3　Queue 对象

Queue 模块（在 Python 3 中为 queue 模块）实现了多生产者-多消费者队列，尤其适合需要在多个线程之间进行信息交换的场合，该模块的 Queue 对象实现了多线程编程所需要的所有锁语义。下面的 Python 2.7.8 代码演示了 Queue 对象的用法。

```
import threading
import time
import Queue #queue in Python3
```

```
class Producer(threading.Thread):
    def __init__(self, threadname):
        threading.Thread.__init__(self, name = threadname)
    def run(self):
        global myqueue
        myqueue.put(self.getName())
        print self.getName(), ' put ', self.getName(), ' to queue.'

class Consumer(threading.Thread):
    def __init__(self, threadname):
        threading.Thread.__init__(self, name = threadname)
    def run(self):
        global myqueue
        print self.getName(), ' get ', myqueue.get(), ' from queue.'
myqueue = Queue.Queue()
plist = []
clist = []

for i in range(10):
    p = Producer('Producer' + str(i))
    plist.append(p)
    c = Consumer('Consumer' + str(i))
    clist.append(c)

for i in plist:
    i.start()
    i.join()
for i in clist:
    i.start()
    i.join()
```

上面的程序运行结果如下：

```
Producer0 put Producer0 to queue.
Producer1 put Producer1 to queue.
Producer2 put Producer2 to queue.
Producer3 put Producer3 to queue.
Producer4 put Producer4 to queue.
Producer5 put Producer5 to queue.
Producer6 put Producer6 to queue.
Producer7 put Producer7 to queue.
Producer8 put Producer8 to queue.
Producer9 put Producer9 to queue.
Consumer0 get Producer0 from queue.
Consumer1 get Producer1 from queue.
```

```
Consumer2 get Producer2 from queue.
Consumer3 get Producer3 from queue.
Consumer4 get Producer4 from queue.
Consumer5 get Producer5 from queue.
Consumer6 get Producer6 from queue.
Consumer7 get Producer7 from queue.
Consumer8 get Producer8 from queue.
Consumer9 get Producer9 from queue.
```

13.3.4 Event 对象

Event 对象是一种简单的线程通信技术，一个线程设置 Event 对象，另一个线程等待 Event 对象。Event 对象的 set()方法可以设置 Event 对象内部的信号标志为真；clear()方法可以清除 Event 对象内部的信号标志，将其设置为假；isSet()方法用来判断其内部信号标志的状态；wait()方法只有在其内部信号状态为真时将很快地执行并返回，若 Event 对象的内部信号标志为假，wait()方法将一直等待至超时或内部信号状态为真。

下面的代码演示了 Event 对象的用法：

```
import threading
class mythread(threading.Thread):
    def __init__(self, threadname):
    threading.Thread.__init__(self, name =threadname)
    def run(self):
        global myevent
        if myevent.isSet():
            myevent.clear()
            myevent.wait()
            print self.getName()
        else:
            print self.getName()
            myevent.set()

myevent =threading.Event()
myevent.set()
tl = []
for i in range(10):
    t =mythread(str(i))
    tl.append(t)

for i in tl:
    i.start()
```

将上面的代码保存为 ThreadSynchronizationUsingEvent.py 文件并运行，您会发现每次的运行结果略有不同，图 13-4 是其中一次的运行结果。

```
D:\教学课件\Python\code\多线程编程>threadsynchronizationusingevent.py
1
 2
 0
4
36

5
8
9
 7
```

图 13-4 使用 Event 对象实现线程同步

本章知识精要

(1) 线程数量并不是越多越好。

(2) threading 模块是 Python 支持多线程编程的重要模块，提供了 Thread、Lock、RLock、Condition、Event、Timer、Semaphore 等大量类和对象。

(3) Thread 类支持两种方法来创建线程：一种方法是为其构造函数传递一个可调用对象；另一种方法是继承 Thread 类并在派生类中重写__init__()和 run()方法。

(4) 创建了线程对象之后，可以调用其 start()方法来启动该线程，join()方法用来等待线程结束或超时。

(5) 可以通过设置线程的 daemon 属性来决定主线程结束时是否需要等待子线程结束，daemon 属性的值默认为 False，即主线程结束时检查并等待子线程结束，如果需要修改 daemon 属性的值则必须在调用 start()方法之前进行修改。

(6) 除了 threading 模块中提供的线程同步对象之外，Queue 或 queue 模块也实现了多生产者-多消费者队列，尤其适合需要在多个线程之间进行信息交换的场合。

习　题

1. 叙述创建线程的方法。
2. 叙述 Thread 对象的方法。
3. 叙述线程对象的 daemon 属性的作用和影响。
4. 解释至少 3 种线程同步方法。

第 14 章　数据库编程

毫无疑问，数据库技术的发展为各行各业都带来了很大方便，数据库不仅支持各类数据的长期保存，更重要的是支持各种跨平台、跨地域的数据查询、共享以及修改，极大方便了人类生活和工作。金融行业、聊天系统、各类网站、办公自动化系统、各种管理信息系统等，都少不了数据库技术的支持。本章主要介绍 SQLite、Access、MySQL、MS SQL Server 等几种数据库的 Python 接口，并通过几个示例来演示数据的增、删、改、查等操作。为了节省篇幅，本章并没有详细介绍数据库的原理、概念以及 SQL 语句的语法，而是假设读者已经了解或者可以通过查阅相关资料学习这部分内容。

14.1　SQLite 应用

SQLite 是内嵌在 Python 中的轻量级、基于磁盘文件的数据库管理系统，不需要服务器进程，支持使用 SQL 语句来访问数据库。该数据库使用 C 语言开发，支持大多数 SQL91 标准，支持原子的、一致的、独立的和持久的事务，不支持外键限制；通过数据库级的独占性和共享锁定来实现独立事务，当多个线程同时访问同一个数据库并试图写入数据时，每一时刻只有一个线程可以写入数据。

SQLite 支持 2TB 大小的单个数据库，每个数据库完全存储在单个磁盘文件中，以 B^+ 树数据结构的形式存储，一个数据库就是一个文件，通过简单复制即可实现数据库的备份。如果需要使用可视化管理工具，请下载并使用 SQLiteManager、SQLite Database Browser 或其他类似工具。如果使用 Python 程序读取 SQLite 记录时显示乱码，可以尝试修改程序并使用 UTF-8 编码格式。

访问和操作 SQLite 数据时，需要首先导入 sqlite3 模块，然后就可以使用其中的功能来操作数据库了，该模块提供了与 DB-API 2.0 规范兼容的 SQL 接口。

使用该模块时，首先需要创建一个与数据库关联的 Connection 对象，例如：

```
import sqlite3
conn = sqlite3.connect('example.db')
```

成功创建 Connection 对象以后，再创建一个 Cursor 对象，并且调用 Cursor 对象的 execute()方法来执行 SQL 语句创建数据表以及查询、插入、修改或删除数据库中的数据，例如：

```
c= conn.cursor()
#创建表
c.execute("CREATE TABLE stocks (date text, trans text, symbol text, qty real,
price real)")
#插入一条记录
c.execute("INSERT INTO stocks VALUES ('2006-01-05','BUY','RHAT',100,35.14)")
```

```
#提交当前事务,保存数据
conn.commit()
#关闭数据库连接
conn.close()
```

如果需要查询表中内容,那么重新创建 Connection 对象和 Cursor 对象之后,可以使用下面的代码来查询:

```
>>> for row in c.execute('SELECT * FROM stocks ORDER BY price'):
    print(row)
```

接下来重点介绍 sqlite3 模块中的 Connection、Cursor 和 Row 等对象。

14.1.1 Connection 对象

Connection 是 sqlite3 模块中最基本也是最重要的一个类,其主要方法如表 14-1 所示。

表 14-1 Connection 对象主要方法

方　　法	说　　明
sqlite3.Connection.execute(sql[, parameters])	执行一条 SQL 语句
sqlite3.Connection.executemany(sql[, parameters])	执行多条 SQL 语句
sqlite3.Connection.cursor()	返回连接的游标
sqlite3.Connection.commit()	提交当前事务,如果不提交的话,那么自上次调用 commit()方法之后的所有修改都不会真正保存到数据库中
sqlite3.Connection.rollback()	撤销当前事务,将数据库恢复至上次调用 commit()方法后的状态
sqlite3.Connection.close()	关闭数据库连接
sqlite3.Connection.create_function(name, num_params, func)	创建可在 SQL 语句中调用的函数,其中 name 为函数名,num_params 表示该函数可以接收的参数个数,func 表示 Python 可调用对象

Connection 对象的其他几个函数都比较容易理解,下面的代码演示了如何在 sqlite3 连接中创建并调用自定义函数:

```
import sqlite3
import hashlib

def md5sum(t):
    return hashlib.md5(t).hexdigest()

con = sqlite3.connect(":memory:")
con.create_function("md5", 1, md5sum)
cur = con.cursor()
cur.execute("select md5(?)", (b"foo",))        #在 SQL 语句中调用自定义函数
print(cur.fetchone()[0])
```

14.1.2 Cursor 对象

Cursor 也是 sqlite3 模块中比较重要的一个对象，该对象具有如下常用方法。

1. execute(sql[, parameters])

该方法用于执行一条 SQL 语句，下面的代码演示了该方法的用法，以及为 SQL 语句传递参数的两种方法，分别使用问号和命名变量作为占位符。

```
import sqlite3

con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table people (name_last, age)")
who = "Dong"
age = 38
#使用问号作为占位符
cur.execute("insert into people values (?, ?)", (who, age))
#使用命名变量作为占位符
cur.execute("select * from people where name_last=:who and age=:age", {"who": who, "age":
age})
print(cur.fetchone())
```

运行结果如图 14-1 所示。

```
>>> ================================ RESTART ====
>>>
(u'Dong', 38)
>>>
```

图 14-1 运行结果

2. executemany(sql, seq_of_parameters)

该方法用来对所有给定参数执行同一个 SQL 语句，该参数序列可以使用不同的方式产生，例如，下面的代码使用迭代来产生参数序列：

```
import sqlite3

class IterChars:
    def __init__(self):
        self.count = ord('a')
    def __iter__(self):
        return self
    def __next__(self):
        if self.count > ord('z'):
            raise StopIteration
        self.count += 1
        return (chr(self.count - 1),) # this is a 1-tuple

con = sqlite3.connect(":memory:")
```

```
cur = con.cursor()
cur.execute("create table characters(c)")
theIter = IterChars()
cur.executemany("insert into characters(c) values (?)", theIter)
cur.execute("select c from characters")
print(cur.fetchall())
```

下面的代码则使用了更为简洁的生成器来产生参数：

```
import sqlite3
import string

def char_generator():
    for c in string.ascii_lowercase:
        yield (c,)
con = sqlite3.connect(":memory:")
cur = con.cursor()
cur.execute("create table characters(c)")
cur.executemany("insert into characters(c) values (?)", char_generator())
cur.execute("select c from characters")
print(cur.fetchall())
```

下面的代码则使用直接创建的序列作为 SQL 语句的参数：

```
import sqlite3

persons = [
    ("Hugo", "Boss"),
    ("Calvin", "Klein")
    ]
con = sqlite3.connect(":memory:")
#创建表
con.execute("create table person(firstname, lastname)")
#插入数据
con.executemany("insert into person(firstname, lastname) values (?, ?)", persons)
#显示数据
for row in con.execute("select firstname, lastname from person"):
    print(row)
print("I just deleted", con.execute("delete from person").rowcount, "rows")
```

运行结果如图 14-2 所示。

3. fetchone()、fetchmany(size=cursor.arraysize)、fetchall()

这 3 个方法用来读取数据。假设数据库通过下面的代码创建并插入数据：

```
import sqlite3
conn = sqlite3.connect("D:/addressBook.db")
cur = conn.cursor()
cur.execute("insert into addressList(name , sex , phon , QQ , address) values
```

```
>>> ================================ RESTART ==
>>>
(u'Hugo', u'Boss')
(u'Calvin', u'Klein')
I just deleted 2 rows
>>>
```

图 14-2 运行结果

```
('王小丫', '女', '13888997011', '66735', '北京市')")
cur.execute("insert into addressList(name, sex, phon, QQ, address) values
('李莉', '女', '15808066055', '675797', '天津市')")
cur.execute("insert into addressList(name, sex, phon, QQ, address) values
('李星草', '男', '15912108090', '3232099', '昆明市')")
conn.commit()
conn.close()
```

则下面的代码演示了使用 fetchall()读取数据的方法：

```
import sqlite3
conn=sqlite3.connect('D:/addressBook.db')
cur=conn.cursor()
cur.execute('select * from addressList')
li=cur.fetchall()
for line in li:
    for item in line:
        if type(item)!=unicode:
            s=str(item)
        else:
            s=item
        print s+'\t',
    print
conn.close()
```

14.1.3 Row 对象

下面通过一个示例来演示 Row 对象的用法，假设数据以下面的方式创建并插入数据：

```
conn = sqlite3.connect(":memory:")
c = conn.cursor()
c.execute("create table stocks(date text, trans text, symbol text, qty real,
price real)")
c.execute("""insert into stocks values ('2006-01-05','BUY','RHAT',100,35.14)""")
conn.commit()
c.close()
```

那么，可以使用下面的方式来读取其中的数据：

```
>>> conn.row_factory = sqlite3.Row
```

```
>>> c = conn.cursor()
>>> c.execute('select * from stocks')
<sqlite3.Cursor object at 0x7f4e7dd8fa80>
>>> r = c.fetchone()
>>> type(r)
<class 'sqlite3.Row'>
>>> tuple(r)
('2006-01-05', 'BUY', 'RHAT', 100.0, 35.14)
>>> len(r)
5
>>> r[2]
'RHAT'
>>> r.keys()
['date', 'trans', 'symbol', 'qty', 'price']
>>> r['qty']
100.0
>>> for member in r:
        print(member)
2006-01-05
BUY
RHAT
100.0
35.14
```

14.2 访问其他类型数据库

除了 SQLite 数据库以外，Python 还可以操作 Access、MS SQL Server 和 MySQL 等多种类型的数据库，本节中对几种常见的接口逐一进行简单介绍。

14.2.1 操作 Access 数据库

首先需要安装 Python for Windows extensions，即 pywin32。然后可以参考下面的步骤和方式来访问 Access 数据库。

1. 建立数据库连接

```
import win32com.client
conn = win32com.client.Dispatch(r'ADODB.Connection')
DSN = 'PROVIDER=Microsoft.Jet.OLEDB.4.0;DATA SOURCE=C:/MyDB.mdb;'
conn.Open(DSN)
```

2. 打开记录集

```
rs = win32com.client.Dispatch(r'ADODB.Recordset')
rs_name = 'MyRecordset'      # 表名
rs.Open('[' + rs_name + ']', conn, 1, 3)
```

3. 操作记录集

```
rs.AddNew()
rs.Fields.Item(1).Value = 'data'
rs.Update()
```

4. 操作数据

```
conn = win32com.client.Dispatch(r'ADODB.Connection')
DSN = 'PROVIDER=Microsoft.Jet.OLEDB.4.0;DATA SOURCE=C:/MyDB.mdb;'
sql_statement = "insert into [Table_Name] ([Field_1], [Field_2]) values ('data1',
'data2')"
conn.Open(DSN)
conn.Execute(sql_statement)
conn.Close()
```

5. 遍历记录

```
rs.MoveFirst()
count = 0
while 1:
    if rs.EOF:
        break
    else:
        count = count + 1
    rs.MoveNext()
```

在操作 Access 数据库时,如果一个记录集是空的,那么将指针移动到第一个记录将导致一个错误,因为此时 recordcount 是无效的。解决的方法:打开一个记录集之前,先将 Cursorlocation 设置为 3,然后再打开记录集,此时 recordcount 将是有效的。

```
rs.Cursorlocation = 3
rs.Open('select * from [Table_Name]', conn)        #确保 conn 处于打开状态
rs.RecordCount
```

14.2.2 操作 MS SQL Server 数据库

可以使用 pywin32 和 pymssql 两种不同的方式来访问 MS SQL Server 数据库。

先来了解一下 pywin32 模块访问 MS SQL Server 数据库的步骤。

1. 添加引用

```
import adodbapi
adodbapi.adodbapi.verbose = False # adds details to the sample printout
import adodbapi.ado_consts as adc
```

2. 创建连接

```
Cfg = {'server':'192.168.29.86\\eclexpress','password':'xxxx','db':'pscitemp'}
constr = r"Provider= SQLOLEDB.1; Initial Catalog= %s; Data Source= %s; user ID= %s; Password= %
```

```
s; "%(Cfg['db'], Cfg['server'], 'sa', Cfg['password'])
conn =adodbapi.connect(constr)
```

3. 执行 sql 语句

```
cur =conn.cursor()
sql ="select * from softextBook where title='{0}' and remark3!='{1}'".
format(bookName,flag)
cur.execute(sql)
data =cur.fetchall()
cur.close()
```

4. 执行存储过程

```
#假设 proName 有 3 个参数,最后一个参数传了 None
ret =cur.callproc('procName',(parm1,parm2,None))
conn.commit()
```

5. 关闭连接

```
conn.close()
```

接下来再通过一个示例来简单了解一下使用 pymssql 模块访问 MS SQL Server 数据库的方法。

```
import pymssql
conn =pymssql.connect(host='SQL01', user='user', password='password', database='
mydatabase')
cur =conn.cursor()
cur.execute('create table persons(id INT, name VARCHAR(100))')
cur.executemany("insert into persons values(%d, xinos.king)", [ (1, 'John Doe'), (2, 'Jane Doe
') ])
conn.commit()
cur.execute('select * from persons where salesrep=xinos.king', 'John Doe')
row =cur.fetchone()
while row:
    print "ID=%d, Name=xinos.king" %(row[0], row[1])
    row =cur.fetchone()
cur.execute("select * from persons where salesrep like 'J%'")
conn.close()
```

14.2.3 操作 MySQL 数据库

Python 访问 MySQL 数据库可以使用 MySQLDb 模块,该模块的主要方法有 10 个。

(1) commit():提交事务。

(2) rollback():回滚事务。

(3) callproc(self, procname, args):用来执行存储过程,接收的参数为存储过程名和参数列表,返回值为受影响的行数。

(4) execute(self, query, args):执行单条 SQL 语句,接收的参数为 SQL 语句本身和使用的参数列表,返回值为受影响的行数。

(5) executemany(self, query, args):执行单条 SQL 语句,但是重复执行参数列表里的参数,返回值为受影响的行数。

(6) nextset(self):移动到下一个结果集。

(7) fetchall(self):接收全部的返回结果行。

(8) fetchmany(self, size=None):接收 size 条返回结果行,如果 size 的值大于返回的结果行的数量,则会返回 cursor.arraysize 条数据。

(9) fetchone(self):返回一条结果行。

(10) scroll(self, value, mode='relative'):移动指针到某一行,如果 mode='relative',则表示从当前所在行移动 value 条;如果 mode='absolute',则表示从结果集的第一行移动 value 条。

使用该模块查询 MySQL 数据库记录的方法如下:

```
import MySQLdb
try:
    conn=MySQLdb.connect(host='localhost',user='root',passwd='root',
    db='test',port=3306)
    cur=conn.cursor()
    cur.execute('select * from user')
    cur.close()
    conn.close()
except MySQLdb.Error,e:
    print "Mysql Error %d: %s" %(e.args[0], e.args[1])
```

插入数据的用法如下:

```
import MySQLdb
try:
    conn=MySQLdb.connect(host='localhost',user='root',passwd='root',port=3306)
    cur=conn.cursor()
    cur.execute('create database if not exists python')
    conn.select_db('python')
    cur.execute('create table test(id int,info varchar(20))')
    value=[1,'hi rollen']
    cur.execute('insert into test values(%s,%s)',value)
    values=[]
    for i in range(20):
        values.append((i,'hi rollen'+str(i)))
    cur.executemany('insert into test values(%s,%s)',values)
    cur.execute('update test set info="I am rollen" where id=3')
    conn.commit()
    cur.close()
    conn.close()
```

```
except MySQLdb.Error,e:
    print "MySQL Error %d: %s" %(e.args[0], e.args[1])
```

本章知识精要

（1）SQLite 是内嵌在 Python 中的轻量级、基于磁盘文件的数据库管理系统，不需要服务器进程，支持使用 SQL 语句来访问数据库。

（2）访问和操作 SQLite 数据库时，需要首先导入 sqlite3 模块。

（3）可以使用 pywin32 模块来操作 Access 数据库和 MS SQL Server 数据库，也可以使用 pymssql 模块来操作 MS SQL Server 数据库。

（4）可以使用 MySQLDb 模块操作 MySQL 数据库。

习　题

1. 简单介绍 SQLite 数据库。
2. 使用 Python 内置函数 dir()查看 Cursor 对象中的方法，并使用内置函数 help()查看其用法。
3. 叙述使用 Python 操作 Access 数据库的步骤。
4. 叙述使用 Python 操作 MS SQL Server 数据库的步骤。
5. 叙述 MySQLDb 模块提供的数据库访问方法。

第 15 章　多媒体编程

在本章中，主要介绍图形编程、图像编程、音乐编程以及声音处理与语音识别等模块和技术。本章中用到的大部分模块不是默认安装的，请根据需要下载并安装相应的模块。

15.1　图形编程

计算机图形学主要研究如何使用计算机来生成具有真实感的图形，涉及的内容主要包括三维建模、图形变换、光照模型、纹理映射和阴影模型等内容，在机械制造、虚拟现实、游戏开发、漫游系统设计、产品展示等多个领域具有重要的应用。随着 3D 打印机的诞生，只要有模型就能够快速生成实物，这无疑会大大扩展计算机图形学的应用范围，例如，可以使用计算机图形学制作出各种可爱的模型，然后参照这些模型使用 3D 打印机批量生产各种食品、玩偶和饰品等。目前大部分计算机图形学的书籍都是基于 OpenGL 的，Python 也提供了 PyOpenGL，这极大地方便了编写图形学程序的 Python 程序员。

15.1.1　创建图形编程框架

Python 的跨平台扩展模块 PyOpenGL 封装了 OpenGL API，支持图形编程所需要的所有功能。使用该模块进行图形编程的步骤如下。

（1）导入模块。

```
from OpenGL.GL import *
from OpenGL.GLU import *
from OpenGL.GLUT import *
import sys
```

（2）使用 OpenGL 创建窗口类。

```
class MyPyOpenGLTest:
```

（3）重写构造函数，初始化 OpenGL 环境，指定显示模式以及用于绘图的函数。

```
def __init__(self, width = 640, height = 480, title = 'MyPyOpenGLTest'):
    glutInit(sys.argv)
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH)
    glutInitWindowSize(width, height)
    self.window = glutCreateWindow(title)
    glutDisplayFunc(self.Draw)
    glutIdleFunc(self.Draw)
    self.InitGL(width, height)
```

(4) 根据特定的需要,进一步完成 OpenGL 的初始化。

```
def InitGL(self, width, height):
    glClearColor(0.0, 0.0, 0.0, 0.0)
    glClearDepth(1.0)
    glDepthFunc(GL_LESS)
    glShadeModel(GL_SMOOTH)
    glEnable(GL_POINT_SMOOTH)
    glEnable(GL_LINE_SMOOTH)
    glEnable(GL_POLYGON_SMOOTH)
    glMatrixMode(GL_PROJECTION)
    glHint(GL_POINT_SMOOTH_HINT,GL_NICEST)
    glHint(GL_LINE_SMOOTH_HINT,GL_NICEST)
    glHint(GL_POLYGON_SMOOTH_HINT,GL_FASTEST)
    glLoadIdentity()
    gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)
    glMatrixMode(GL_MODELVIEW)
```

(5) 定义自己的绘图函数。

```
def Draw(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glutSwapBuffers()
```

(6) 消息主循环。

```
def MainLoop(self):
    glutMainLoop()
```

(7) 实例化窗口类,运行程序。

```
if __name__=='__main__':
    w=MyPyOpenGLTest()
    w.MainLoop()
```

15.1.2 绘制文字

可以使用 glutBitmapCharacter()函数在绘图窗口上绘制文字,该函数每次只能绘制一个字符,可以使用循环结构来输出多个字符。改写前面图形编程框架中的 Draw()函数,就可以实现该功能。

```
def Draw(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glColor3f(1.0, 1.0, 1.0)
    glTranslatef(0.0, 0.0, -1.0)
    glRasterPos2f(0.0, 0.0)
```

```
s ='PyOpenGL is the binding layer between Python and OpenGL.'
for ch in s:
    glutBitmapCharacter(GLUT_BITMAP_8_BY_13, ord(ch))
```

15.1.3 绘制图形

在 OpenGL 中绘制图形的代码需要放在 glBegin(mode)和 glEnd()这一对函数的调用之间,其中 mode 表示绘图类型,取值范围如表 15-1 所示。

表 15-1 mode 取值

取　　值	说　　明	取　　值	说　　明
GL_POINTS	绘制点	GL_TRIANGLE_STRIP	绘制三角形串
GL_LINES	绘制直线	GL_TRIANGLE_FAN	绘制三角扇形
GL_LINE_STRIP	绘制连续直线,不封闭	GL_QUADS	绘制四边形
GL_LINE_LOOP	绘制封闭的连续直线	GL_QUAD_STRIP	绘制四边形串
GL_TRIANGLES	绘制三角形	GL_POLYGON	绘制多边形

例如,将前面给出的图形编程框架中的 Draw()函数改写成下面的代码,则可以绘制一个彩色三角形和一条彩色直线。在这段代码中,首先设置绘制模式为多边形,然后依次绘制该多边形的顶点,绘制每个顶点之前设置顶点颜色,最后修改绘制模式为直线并指定直线段的端点颜色和位置。需要注意的是,使用 glColor3f()函数设置颜色之后,直到下一次使用该函数改变颜色之前,绘制的所有顶点都使用这个颜色。或者说,OpenGL 采用的是"状态机"工作方式,一旦设置了某种状态之后,除非显式修改该状态,否则该状态将一直保持。

```
def Draw(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslatef(-2.0, 0.0, -8.0)
    #绘制二维图形,z 坐标为 0
    glBegin(GL_POLYGON)                #绘制多边形
    glColor3f(1.0, 0.0, 0.0)           #4 设置顶点颜色
    glVertex3f(0.0, 1.0, 0.0)          #绘制多边形顶点
    glColor3f(0.0, 1.0, 0.0)
    glVertex3f(1.0, -1.0, 0.0)
    glColor3f(0.0, 0.0, 1.0)
    glVertex3f(-1.0, -1.0, 0.0)
    glEnd()
    glTranslatef(2.5, 0.0, 0.0)
    #绘制三维图形
    glBegin(GL_LINES)                  #绘制直线
    glColor3f(1.0, 0.0, 0.0)
    glVertex3f(1.0, 1.0, -1.0)
    glColor3f(0.0, 1.0, 0.0)
```

```
    glVertex3f(-1.0, -1.0, 3.0)
    glEnd()
    glutSwapBuffers()
```

上面的代码运行结果如图 15-1 所示。



图 15-1 绘制图形

15.1.4 纹理映射

在现实中,人们主要通过物体表面丰富的纹理细节来区分具有相同形状的不同物体。在三维建模时也往往通过纹理映射来简化建模的工作量,可以在保证图形具有较强真实感的前提下大幅度提高渲染效率。

简单地说,纹理映射就是为物体表面进行贴图以使其呈现出特定的视觉效果。这需要首先准备好纹理,然后构建物体空间坐标和纹理坐标之间的对应关系来完成贴图。可以使用函数来生成一些规则的纹理,例如粗布纹理、棋盘纹理等,也可以将拍摄或通过网络搜索下载的图片作为纹理映射到物体表面上。进行纹理映射之前,首先要读取并设置纹理数据。在前面给出的图形编程框架中增加如下函数用来读取和设置纹理数据:

```
def LoadTexture(self):
    img = Image.open('sample.bmp')
    width, height = img.size
    img = img.tostring('raw', 'RGBX', 0, -1)
    glBindTexture(GL_TEXTURE_2D, glGenTextures(1))
    glPixelStorei(GL_UNPACK_ALIGNMENT,1)
    glTexImage2D(GL_TEXTURE_2D, 0, 4, width, height, 0, GL_RGBA, GL_UNSIGNED_BYTE, img)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)
```

```
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST)
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST)
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL)
```

然后修改图形编程框架中的初始化函数，设置纹理映射属性，并进行背面剔除，修改后的代码如下：

```
def InitGL(self, width, height):
    self.LoadTexture()
    glEnable(GL_TEXTURE_2D)
    glClearColor(0.0, 0.0, 0.0, 0.0)
    glClearDepth(1.0)
    glDepthFunc(GL_LESS)
    glShadeModel(GL_SMOOTH)
    glEnable(GL_CULL_FACE)
    glCullFace(GL_BACK)              #背面剔除
    glEnable(GL_LINE_SMOOTH)
    glEnable(GL_POLYGON_SMOOTH)
    glMatrixMode(GL_PROJECTION)
    glHint(GL_LINE_SMOOTH_HINT,GL_NICEST)
    glHint(GL_POLYGON_SMOOTH_HINT,GL_FASTEST)
    glLoadIdentity()
    gluPerspective(45.0, float(width)/float(height), 0.1, 100.0)
    glMatrixMode(GL_MODELVIEW)
```

接下来，修改图形编程框架中的 Draw()函数，绘制立方体盒子，并使用上面代码读取到的纹理数据进行表面映射：

```
def Draw(self):
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)
    glLoadIdentity()
    glTranslate(0.0, 0.0, -9.0)
    glRotatef(self.x, 1.0, 0.0, 0.0)
    glRotatef(self.y, 0.0, 1.0, 0.0)
    glRotatef(self.z, 0.0, 0.0, 1.0)

    #依次绘制立方体的 6 个面并进行纹理映射
    glBegin(GL_QUADS)
    glTexCoord2f(0.0, 0.0)           #设置纹理坐标
    glVertex3f(-1.0, -1.0, 1.0)      #指定顶点位置
    glTexCoord2f(1.0, 0.0)
    glVertex3f(1.0, -1.0, 1.0)
    glTexCoord2f(1.0, 1.0)
    glVertex3f(1.0, 1.0, 1.0)
    glTexCoord2f(0.0, 1.0)
```

```
glVertex3f(-1.0, 1.0, 1.0)

glTexCoord2f(1.0, 0.0)
glVertex3f(-1.0, -1.0, -1.0)
glTexCoord2f(1.0, 1.0)
glVertex3f(-1.0, 1.0, -1.0)
glTexCoord2f(0.0, 1.0)
glVertex3f(1.0, 1.0, -1.0)
glTexCoord2f(0.0, 0.0)
glVertex3f(1.0, -1.0, -1.0)

glTexCoord2f(0.0, 1.0)
glVertex3f(-1.0, 1.0, -1.0)
glTexCoord2f(0.0, 0.0)
glVertex3f(-1.0, 1.0, 1.0)
glTexCoord2f(1.0, 0.0)
glVertex3f(1.0, 1.0, 1.0)
glTexCoord2f(1.0, 1.0)
glVertex3f(1.0, 1.0, -1.0)

glTexCoord2f(1.0, 1.0)
glVertex3f(-1.0, -1.0, -1.0)
glTexCoord2f(0.0, 1.0)
glVertex3f(1.0, -1.0, -1.0)
glTexCoord2f(0.0, 0.0)
glVertex3f(1.0, -1.0, 1.0)
glTexCoord2f(1.0, 0.0)
glVertex3f(-1.0, -1.0, 1.0)

glTexCoord2f(1.0, 0.0)
glVertex3f(1.0, -1.0, -1.0)
glTexCoord2f(1.0, 1.0)
glVertex3f(1.0, 1.0, -1.0)
glTexCoord2f(0.0, 1.0)
glVertex3f(1.0, 1.0, 1.0)
glTexCoord2f(0.0, 0.0)
glVertex3f(1.0, -1.0, 1.0)

glTexCoord2f(0.0, 0.0)
glVertex3f(-1.0, -1.0, -1.0)
glTexCoord2f(1.0, 0.0)
glVertex3f(-1.0, -1.0, 1.0)
glTexCoord2f(1.0, 1.0)
glVertex3f(-1.0, 1.0, 1.0)
glTexCoord2f(0.0, 1.0)
```

```
        glVertex3f(-1.0, 1.0, -1.0)

        glEnd()
        glutSwapBuffers()
```

15.1.5 处理键盘/鼠标事件

如果需要使用鼠标或键盘来操作图形，如平移、旋转和缩放等，那么首先需要在初始化函数中指定接受键盘和鼠标事件的函数，即增加下面两行代码：

```
def __init__(self, width=640, height=480, title='MyPyOpenGLTest'):
    ⋮
    glutKeyboardFunc(self.KeyPress)
    glutMouseFunc(self.Mouse)
    ⋮
```

然后在窗口类中增加下面的函数定义，用来接受并处理键盘和鼠标事件：

```
def Mouse(self, button, mode, x, y):
    if button==GLUT_RIGHT_BUTTON and mode==GLUT_DOWN:
        print 'yes'
def KeyPress(self, key, x, y):
    print key
```

15.2 图像编程

Python Imaging Library (PIL)是支持 Python 的图像处理扩展模块，支持多种图像格式，并提供非常强大的图像处理功能。PIL 模块需要单独进行安装后才能使用，在 PIL 中主要提供 Image、ImageChops、ImageColor、ImageDraw、ImagePath、ImageFile、ImageEnhance 和 PSDraw 以及其他一些模块来支持图像的处理。本书编写时 PIL 模块的官方版本只支持 Python 2，但也有私人编译的 Python 3 版本。

使用该扩展库时，首先需要导入，例如：

```
>>> from PIL import Image
```

接下来，我们通过几个示例来简单演示一下该模块的用法。

(1) 打开图像文件。

```
>>> im=Image.open('sample.jpg')
```

(2) 显示图像。

```
>>> im.show()
```

(3) 查看图像信息。

```
>>> print im.format
JPEG
```

```
>>>print im.size
(360, 468)
```

(4) 查看图像直方图。

```
>>>im.histogram()
```

(5) 读取像素值。

```
>>>print im.getpixel((100,50))
(124, 126, 123)
```

(6) 设置像素值,通过读取和修改图像像素值可以实现图像点运算。

```
>>>im.putpixel((100,50),(128,30,120))          #第二个参数用来指定目标像素的颜色值
```

(7) 保存图像文件。

```
>>>im.save('sample1.jpg')
```

(8) 转换图像格式。

```
>>>im.save('sample.bmp')                       #通过该方法可以进行格式转换
```

(9) 图像缩放。

```
>>>im1 = im.resize((100,100))
```

(10) 旋转图像,rotate()方法支持任意角度的旋转,而 transpose()方法支持部分特殊角度的旋转,如 90°、180°、270°旋转以及水平、垂直翻转等。

```
>>>im2 = im.rotate(90)
>>>im3 = im.transpose(Image.ROTATE_180)        #180°旋转
>>>im4 = im.transpose(Image.FLIP_LEFT_RIGHT)   #水平翻转
```

(11) 图像裁剪与粘贴。

```
>>>box = (120,194,220,294)
>>>region = im.crop(box)                       #定义裁剪区域
>>>region = region.transpose(Image.ROTATE_180)
>>>im.paste(region,box)                        #粘贴
>>>im.show()
```

例如,图 15-2 是原始 lena 图像,而图 15-3 是将其中一部分进行旋转 180°以后的结果,请注意左下角区域图像的变化。

(12) 将彩色图像分离为红、绿、蓝三分量子图,分离后每个图像大小与原图像一样,但是只包含一个颜色分量。

```
>>>r,g,b = im.split()
```

(13) 图像增强。

```
>>>from PIL import ImageFilter
```



图 15-2　原始 lena 图像



图 15-3　部分区域被旋转 180°以后的 lena 图像

```
>>> im5 = im.filter(ImageFilter.DETAIL)
```

（14）图像模糊。

```
>>> im6 = im.filter(ImageFilter.BLUR)
```

（15）图像边缘提取。

```
>>> im7 = im.filter(ImageFilter.FIND_EDGES)
```

（16）图像点运算，整体变暗或变亮。

```
>>> im8 = im.point(lambda i:i * 1.3)
```

```
>>> im9 = im.point(lambda i:i * 0.7)
```

也可使用图像增强模块来实现上面的功能，例如：

```
>>> from PIL import ImageEnhance
>>> enh = ImageEnhance.Brightness(im)
>>> enh.enhance(1.3).show()
```

(17) 图像冷暖色调调整。

```
>>> r,g,b = im.split()
>>> r = r.point(lambda i:i * 1.3)
>>> g = g.point(lambda i:i * 0.9)
>>> b = b.point(lambda i:0)
>>> im10 = Image.merge(im.mode, (r,g,b))
>>> im10.show()
```

(18) 图像对比度增强。

```
>>> im = Image.open('sample.jpg')
>>> im.show()
>>> from PIL import ImageEnhance
>>> enh = ImageEnhance.Contrast(im)
>>> enh.enhance(1.3).show()
```

15.3 音乐编程

pygame 模块的 mixer 模块提供了支持音乐文件播放的功能，可以参考下面的网址下载 pygame 模块并了解 pygame 模块更多的知识和用法。

```
http://pygame.org/ftp/
http://eyehere.net/2011/python-pygame-novice-professional-index/
http://www.pygame.org/docs/ref/
```

pygame 模块除了提供 mixer 模块支持音乐播放之外，还包含了大量其他支持游戏编程的模块，如表 15-2 所示。

表 15-2 pygame 主要模块

模 块	说 明	模 块	说 明
display	屏幕显示	time	时间控制
event	事件处理	cursors	控制鼠标指针
image	图像处理	transform	修改和移动图像
mouse	鼠标消息处理	key	读取键盘按键
movie	视频文件播放，需要安装 PyMedia	font	使用字体
surface	绘制屏幕		

pygame 模块中用于音乐播放有关的方法主要在 mixer 模块中，如表 15-3 所示。

表 15-3　pygame. mixer 的主要方法

方　　法	说　　明
pygame. mixer. init()	初始化，必须最先调用
pygame. mixer. music. load(filename)	打开音乐文件
pygame. mixer. music. play(count, start)	播放音乐文件
pygame. mixer. music. stop()	停止播放
pygame. mixer. music. pause()	暂停播放
pygame. mixer. music. unpause()	继续播放
pygame. mixer. music. get_busy()	检测声卡是否正被占用

另外，跨平台音频/视频播放支持库 Phonon 也提供了播放音频和视频文件的功能，或者也可以使用 DirectSound 或 WMPlayer. ocx 或其他控件进行音乐文件播放。本节主要以 pygame 为例介绍音乐播放器的编写。

下面的代码使用 pygame. mixer 模块编写了一个简单的音乐播放器。程序运行后，将会自动随机播放指定文件夹中所有 mp3 音乐文件，并自动打印显示当前正在播放的音乐文件名。当然，可以修改这段代码以支持其他类型音乐文件的播放，或者还可以结合第 9 章 GUI 编程的知识编写一个更加漂亮美观的音乐播放器程序。

```
import os
import pygame
import random
import time

folder = r'h:\music'
musics = [folder+ '\\'+music for music in os.listdir(folder) if music.endswith('.mp3')]
total = len(musics)
pygame.mixer.init()
while True:
    if not pygame.mixer.music.get_busy():
        nextMusic = random.choice(musics)
        pygame.mixer.music.load(nextMusic)
        pygame.mixer.music.play(1)
        print 'playing…',nextMusic
    else:
        time.sleep(1)
```

15.4　语 音 识 别

使用 Python 编写语音识别程序需要用到 speech 模块，并且需要安装 pywin32 和 Microsoft Speech SDK。

speech 模块支持的主要功能有：文本合成语音，将键盘输入的文本信息以语音信号方

式输出；语音识别，将输入的语音信号识别为文本；特定词的识别，对输入的语音信号进行特定词的捕捉；特定用户、特定词的识别，能够对不同人、不同特定词进行识别。

speech 模块的主要方法如表 15-4 所示。

表 15-4　speech 模块的主要方法

方　法	说　明
speech. say(phrase)	读出给定的文本
speech. input(prompt=None，phraselist=None)	打印信息 prompt 提示用户使用语音录入在 phraselist 中列出的文本，并返回用户录入的内容。该函数会阻塞当前线程直至得到用户录入或者按 Ctrl+C 组合键结束
speech. listenfor(phraselist，callback)	如果用户语音录入 phraselist 中的任何文本，则自动调用回调函数 callback，并返回 Listener 对象
speech. listenforanything(callback)	得到用户语音录入的内容后自动执行回调函数 callback(spoken_text，listener)，并返回 Listener 对象
speech. Listener. islistening(self)	当 Listener 对象处于监听状态时返回 True
speech. Listener. stoplistening(self)	停止监听，当 Listener 对象处于监听状态时返回 True
speech. islistening()	只要有 Listener 对象正在监听则返回 True
speech. stoplistening()	停止所有 Listener 对象的监听状态，如果有 Listener 对象处于监听状态则返回 True

例如，下面的代码让计算机读出用户输入的内容，当用户输入 stop 时结束。

```
>>>while True:
    words =input("Please input some words:")
    if words.lower() =='stop':
        break
    speech.say(words)
```

下面的代码让计算机接受用户语音输入，并重复一遍用户语音录入的内容，以文字形式显示用户语音输入的内容。

```
>>>contents =speech.input()
>>>speech.say(contents)
>>>print contents
```

本章知识精要

(1) Python 的跨平台扩展模块 PyOpenGL 封装了 OpenGL API，支持图形编程所需要的所有功能。

(2) OpenGL 采用的是“状态机”工作方式，一旦设置了某种状态之后，除非显式修改该状态，否则该状态将一直保持，例如图形顶点的颜色、法向量和纹理坐标等。

(3) PIL 是支持 Python 的图像处理模块，提供了强大的图像处理功能。

(4) 可以使用 pygame. mixer、Phonon、DirectSound 或 WMPlayer. ocx 等多种方式进行

音乐文件播放。

(5) 使用 Python 编写语音识别程序需要用到 speech 模块，并且需要安装 pywin32 模块和 Microsoft Speech SDK。

习　　题

1. 编程程序，在窗口上绘制一个三角形，设置 3 个顶点为不同的颜色，并对内部进行光滑着色。

2. 编写程序，读取两幅大小一样的图片，然后将两幅图像的内容叠加到一幅图像，结果图像中每个像素值为原来两幅图像对应位置像素值的平均值。

3. 编写程序，读取一幅图像的内容，将其按象限分为 4 等份，然后 1、3 象限内容交换，2、4 象限内容交换，生成一幅新图像。

4. 结合 GUI 编程知识，编写一个程序，创建一个窗口并在上面放置两个按钮，分别为“开始播放”和“暂停播放”，将本章 15.3 节中的音乐播放程序进行封装。

5. 运行本章 15.4 中的代码并查看运行结果。

第 16 章　逆向工程与软件分析

对于大多数程序员而言，或许并不关心关于硬件与操作系统底层或者软件运行机制的细节，只需要也只希望把更多的精力放在高层的业务逻辑实现上面。但是毫无疑问，如果您对底层细节了解或熟悉的话，就能够对自己开发的软件进行更好的把握和控制。对硬件和系统底层的深刻理解有利于写出更好的应用程序，对于程序员的职业发展也是非常有帮助的。而在某些领域，逆向工程是解决问题非常重要的方式，甚至可能是唯一的方式，例如软件安全测试、加密解密、软件汉化、漏洞挖掘、计算机取证、恶意软件分析和版权保护等。在这些领域中，一般很难获得软件源代码，只能对二进制可执行文件进行分析，而可执行文件由于编译器的优化一般变得非常难以理解，甚至很多恶意软件根本没有可独立运行的文件，而是将代码注入到其他正常进程中。另外，最近几年提出的 ROP、JOP 攻击甚至没有注入任何代码，仅仅通过精心选择和重新组合进程中已有的指令序列就可以实现自己的恶意功能。所有这些都给安全分析人员造成很大困难和挑战，这要求分析人员对逆向工程有着更全面而准确的理解和把握，并且能够熟练运用各种成熟的工具，必要的时候甚至需要自己编写程序来完成分析任务。从另一个角度来讲，从源代码级别对软件进行分析，无法获知编译器对最终可执行文件造成的影响，或者说，很难保证编译器能够忠实地、毫无错误地工作。不幸的是，编译器本身也是软件的一种，同样也有可能存在漏洞。从底层对最终可执行文件进行分析，可以综合考虑各方面的因素(包括加载过程、进程管理和内存管理等)，虽然难度相对较大，但是可以得到更加全面和准确的信息，甚至可以控制和修改软件的运行过程。

在本章中，重点介绍 Windows 平台上 PE 文件的分析。PE 的全称是 Portable Executable，指可移植的可执行文件，目前的最新版本是 2013 年 2 月 6 日发布的 8.3 版。PE 文件包括 exe 文件、com 文件、dll 文件、ocx 文件、sys 文件、scr 文件等 Windows 平台上所有可执行文件类型，可以说 PE 文件是 Windows 操作系统和 Windows 平台上所有软件和程序能够正常运行的重要基础。

需要说明的是，应尽量避免直接在本地物理主机上分析恶意软件，以免被恶意软件感染而造成不必要的损失。为了保证物理主机安全，同时也为了能够在分析环境被恶意软件感染之后快速恢复系统，建议您使用 VirtualBox、VMware、QEMU 等虚拟机系统或沙箱系统进行保护。如果您没有条件使用虚拟机或沙箱系统，也请使用 Deep Freeze、Truman、FPG 或其他类似软件来保护物理主机以防止系统被感染。

16.1　主流项目与插件简介

在软件安全和逆向工程领域，有大量的成熟工具以及针对不同工具和目的开发的各种插件，例如 IDA Pro、OllyDbg、WinDbg、W32DASM、PEid、ssdeep、DiStorm、DisView、LordPE、PIN、Universal PE Unpacker 和 Sample Chart Builder 等，可以说是数不胜数。本书中主要介绍使用 Python 开发或可以使用 Python 进行二次开发的工具和插件，以及如何

使用 Python 开发 PE 文件逆向分析工具。

16.1.1 主流项目

目前已有大量使用 Python 作为主要语言开发的软件逆向分析工具，下面列出了知名度较高的几款。

（1）PyEmu：可编写脚本的模拟器，对恶意软件分析非常有用。

（2）Immunity Debugger：著名的调试器，是在 OllyDbg 的源代码基础上建立起来的，外观与 OllyDbg 非常相似，并且两者共享很多的底层功能和控制。Immunity Debugger 带有内置的 Python 接口和专门用于研究漏洞和执行恶意软件分析的强大 API，是可编写脚本的 GUI 和命令行软件调试器，支持 exploit 编写、二进制可执行文件逆向工程等各种应用。

（3）Paimei：完全使用 Python 编写，是非常成熟的逆向工程框架，包括 PyDBG、PIDA 和 pGRAPH 等多个可扩展模块，可以执行大量静态分析和动态分析，例如模糊测试、代码覆盖率跟踪和数据流跟踪等。

（4）ropper：比较成熟的 ROP Gadgets 查找与可执行文件分析工具，其反汇编部分使用了成熟的 Capstone 框架。

（5）WinAppDbg：纯 Python 调试器，没有本机代码，使用 ctypes 封装了许多与调试器有关的 Win32 API 调用，并且为操作线程、库和进程提供了强有力的抽象。利用该工具可以将自己编写的脚本附加为调试器、跟踪执行、拦截 API 调用，以及在待调试进程中处理事件，并且可以设置各种断点。

（6）YARA：恶意软件识别和分类引擎，也可以利用 YARA 创建规则以检测字符串、入侵序列、正则表达式和字节模式等。既可以使用命令行模式下的 YARA 工具扫描文件，也可以利用 YARA 提供的 API 函数将 YARA 扫描引擎集成到 C 或 Python 语言编写的工具中。

16.1.2 常用插件

经过多年努力，不同的研究人员分别推出了用于不同软件分析需要的 IDA、OllyDbg 以及 Immunity Debugger 插件，大大简化了分析人员的工作。除了以下几种常用插件，还可以通过 SDK 编写自己的插件，或者通过一定技术将 OllyDbg 插件转换为 Immunity Debugger 插件，大大提高了插件的应用范围和生命力。

（1）IDAPython 插件：IDAPython 是运行于交互式反汇编器 IDA 的插件，用于实现 IDA 的 Python 编程接口。IDA 在逆向工程领域具有广泛的应用，尤其是二进制文件静态分析，其强大的反汇编功能一直在业内处于领先水平。IDAPython 插件使得 Python 脚本程序能够在 IDA 中运行并实现自定义的软件分析功能，通过该插件运行的 Python 脚本程序可以访问整个 IDA 数据库，并且可以方便地调用所有 IDC 函数和使用所有已安装的 Python 模块中的功能。目前，IDAPython 还不支持 Python 3，较高版本的 IDA 中集成了 IDAPython 插件，如果需要安装或升级，需要登录其官方网站下载安装适合您当前已安装 Python 和 IDA 版本的 IDAPython 插件。

（2）Hex-Rays Decompiler：IDA 插件，非常成熟的反编译插件。

（3）PatchDiff2：IDA 插件，主要用于补丁对比。

（4）BinDiff：IDA 插件，主要用于二进制文件差异比较。

（5）hidedebug：Immunity Debugger 插件，可以隐藏调试器的存在，用来对抗某些通用的反调试技术。

（6）IDAStealth：IDA 插件，可隐藏 IDA Debugger 的存在，用来对抗某些通用的反调试技术。

16.2 IDAPython 与 Immunity Debugger 编程

16.2.1 IDAPython 编程

安装 IDAPython 插件时，一定要正确选择适合已安装的 Python 和 IDA 版本，否则可能无法在 IDA 中加载 IDAPython 和运行您编写的 Python 程序。安装成功以后，启动 IDA 会看到软件界面最下端有个 Python 标志，在后面的文本框中可以直接输入并运行 Python 代码，如图 16-1 所示。另外，使用 IDLE 或其他 Python 开发环境或者记事本等文本编辑器编写 Python 程序后，在 IDA 主界面中单击菜单 File，然后选择 Script file，在弹出的“脚本文件选择”对话框中，可以看到文件类型为.idc 和.py 两种，如图 16-2 所示。也就是说，现在可以在 IDA 中运行 Python 程序了，最后选择并运行您自己编写的 Python 程序来实现自定义的二进制文件分析任务。

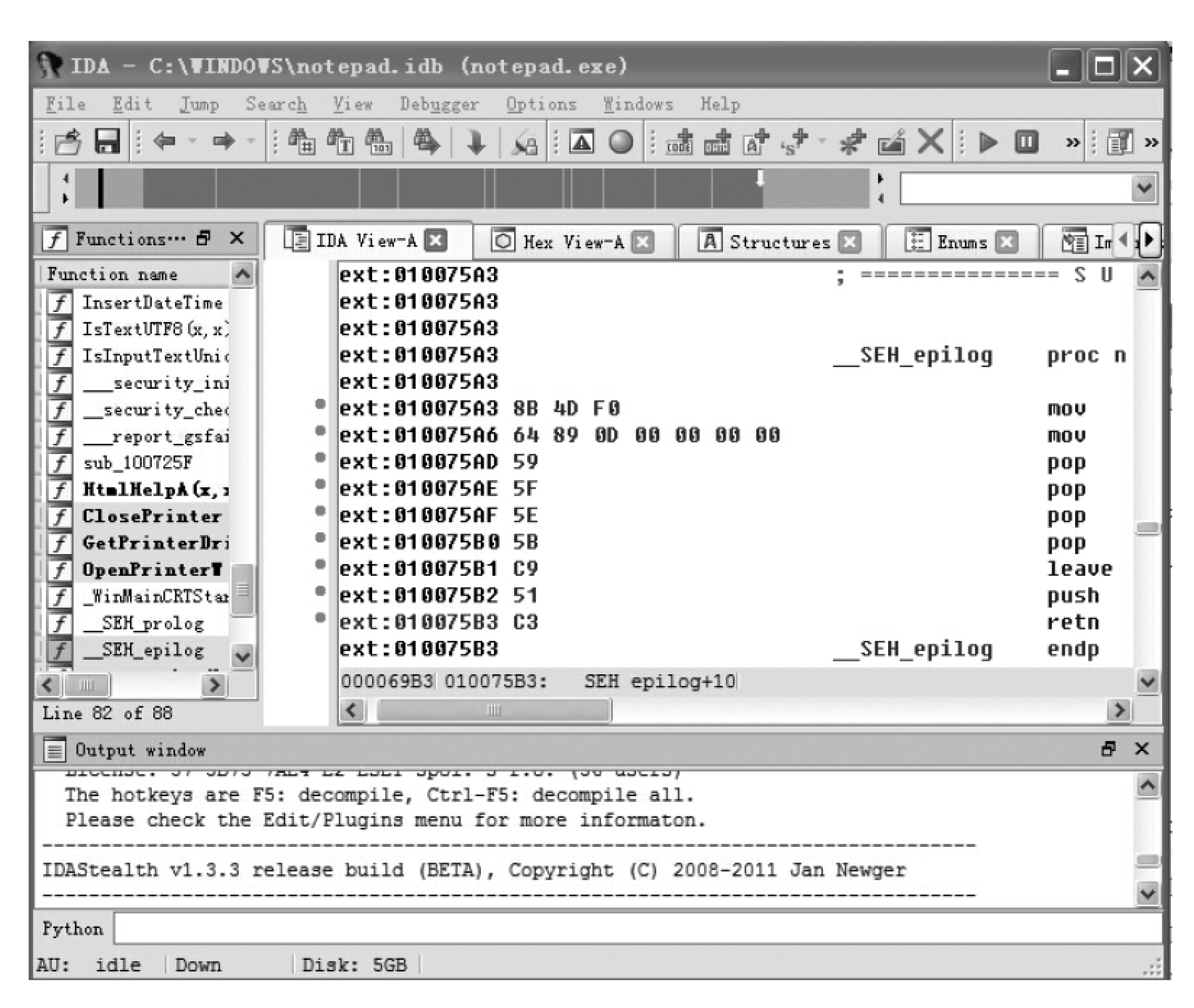


图 16-1 IDAPython 插件安装成功的 IDA 软件界面

接下来，主要通过几个示例来演示如何使用 Python 编程并在 IDA 中运行来实现 PE 文件分析。详细的 IDC 库函数可以查阅官方网址：

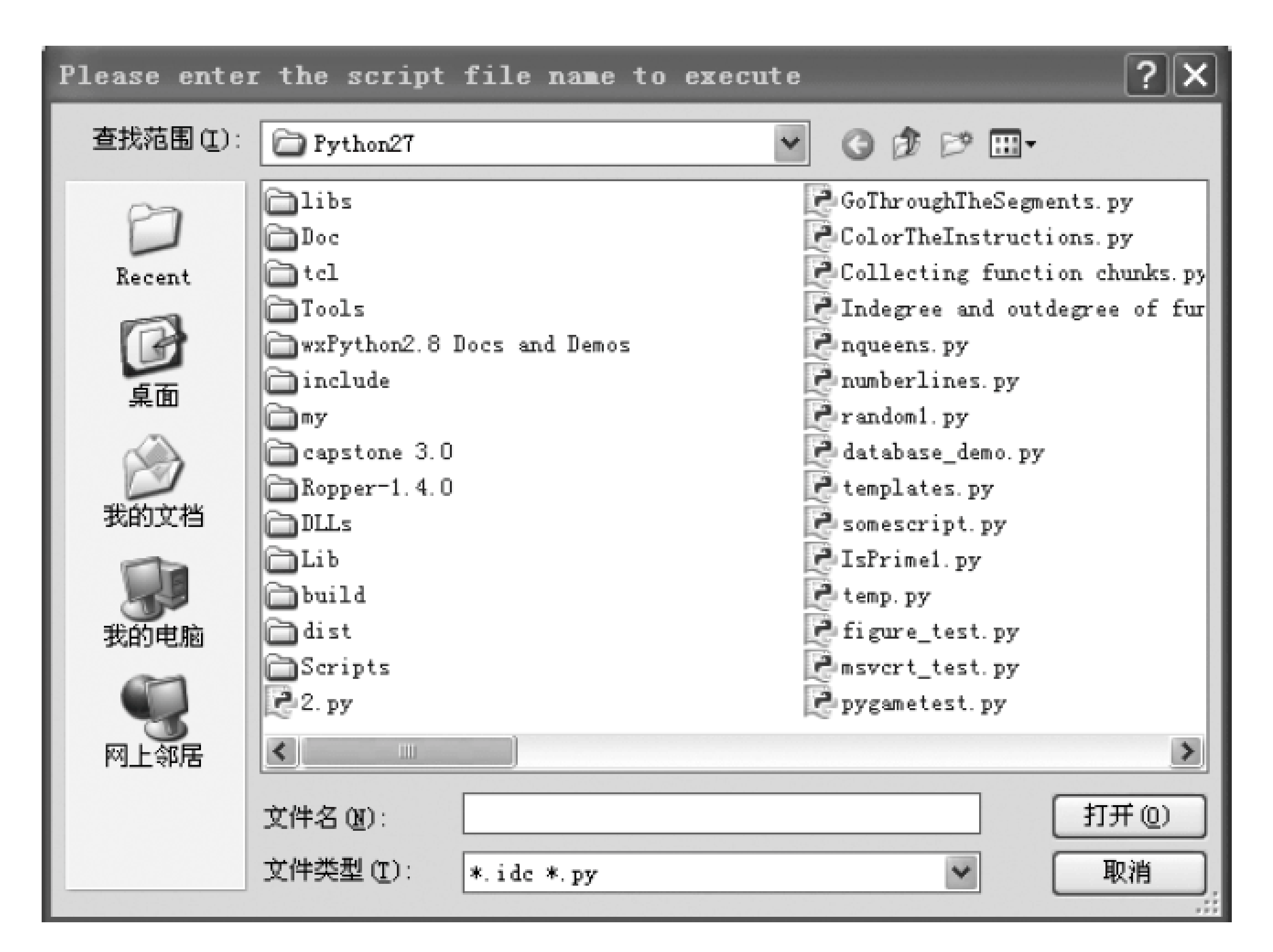


图 16-2 “脚本文件选择”对话框

https://www.hex-rays.com/products/ida/support/idadoc/162.shtml

和

https://www.hex-rays.com/products/ida/support/idapython_docs/index.html

(1) 查看 PE 文件中所有段的名字、起始地址以及结束地址。

```
for seg in Segments():
    print SegName(seg),'(',hex(SegStart(seg)),',',hex(SegEnd(seg)),')'
```

(2) 查看 PE 文件中所有段的名字与长度。

```
segments = dict()
for seg_ea in Segments():
    segments[SegName(seg_ea)] = SegEnd(seg_ea) - seg_ea
for seg_name,seg_data in segments.items():
    print seg_name,seg_data
```

(3) 查看 PE 文件中所有函数信息。

```
for segment in Segments():
    for function_ea in Functions(SegStart(segment),SegEnd(segment)):
        print hex(function_ea),GetFunctionName(function_ea)
```

(4) 查找 PE 文件中指定函数调用,并将该行设置为红色进行高亮显示。

```
from idaapi import *
danger_functions = ['strcpy','sprintf','strncpy','memcpy']
for func in danger_functions:
    addr = LocByName(func)
    if addr != BADADDR:
        cross_refs = CodeRefsTo(addr,0)
```

```
        print 'Cross References to %s'%func
        print '----------------------------'
        for ref in cross_refs:
            print '%08x'%ref
            SetColor(ref,CIC_ITEM,0x0000ff)
        print '----------------------------'
```

（5）遍历函数 chunk。

```
function_chunks = []
for ea in Functions():
    func_iter =idaapi.func_tail_iterator_t(idaapi.get_func(ea))
    status =func_iter.main()
    while status:
        chunk =func_iter.chunk()
        function_chunks.append((chunk.startEA,chunk.endEA))
        status =func_iter.next()
for chunk in function_chunks:
    print (hex(chunk[0]),hex(chunk[1])),'belongs to function:',GetFunctionName(chunk[0])
```

（6）统计函数入度与出度。

```
from sets import Set
ea =ScreenEA()
callers =dict()
callees =dict()
for function_ea in Functions(SegStart(ea),SegEnd(ea)):   #遍历当前段中的函数
    f_name =GetFunctionName(function_ea)                   #获取函数名字
    callers[f_name] =Set(map(GetFunctionName,CodeRefsTo(function_ea,0)))
                                                            #调用该函数的所有函数
    for ref_ea in CodeRefsTo(function_ea,0):      #遍历调用该函数的所有函数
        caller_name =GetFunctionName(ref_ea);
        callees[caller_name] =callees.get(caller_name,Set())
        callees[caller_name].add(f_name)
functions =Set(callees.keys()+callers.keys())
for f in functions:
    print '%-4d::%s::%4d'%(len(callers.get(f,[])),f,len(callees.get(f,[])))
```

（7）统计 PE 文件中的指令频度。

```
mnemonics =dict()
ea =ScreenEA()
for head in Heads(SegStart(ea), SegEnd(ea)):
    if isCode(GetFlags(head)):
        mnem =GetMnem(head)
        mnemonics[mnem] =mnemonics.get(mnem, 0)+1
mnem_list =map(lambda x:(x[1],x[0]), mnemonics.items())
```

```
    mnem_list.sort()
    for cnt, mnem in mnem_list:
        print mnem, cnt
```

针对某动态链接库文件，上面的代码运行结果如图 16-3 所示。

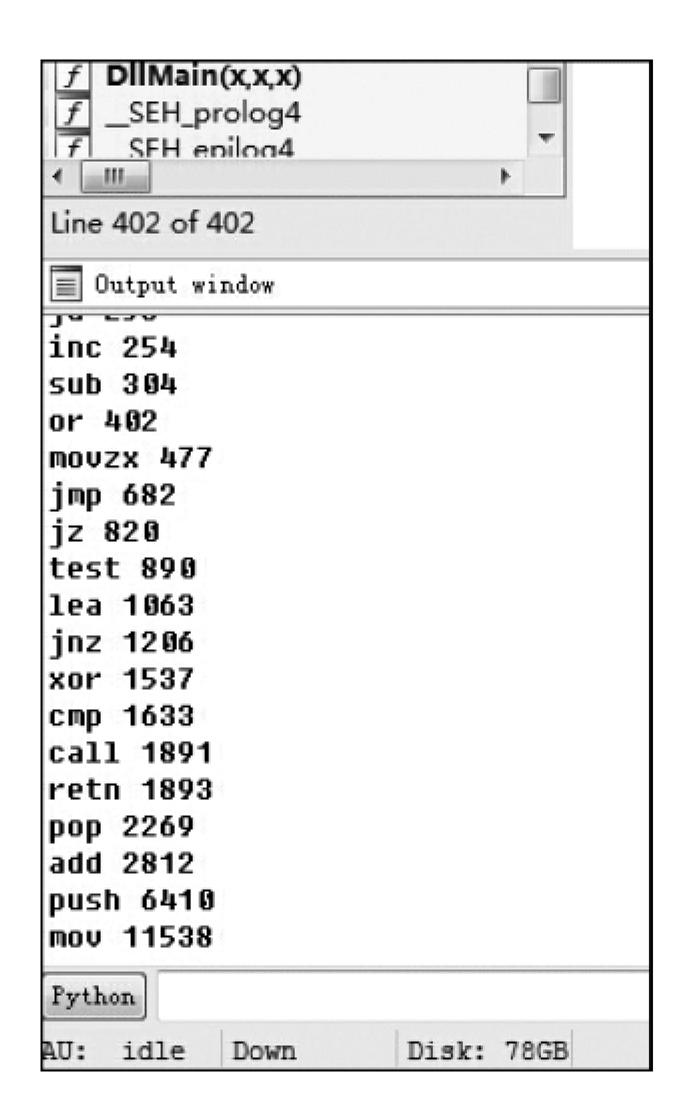


图 16-3　指令频度统计结果

(8)查找潜在的 ROP Gadgets。

ROP(Return-Oriented Programming)是近几年来流行的一种攻击方式。在早些年，黑客通过各种溢出漏洞和保护机制的缺陷来实现任意代码注入和执行，后来由于数据执行保护(Data Execution Prevention，DEP)和地址空间布局随机化(Address Space Layout Randomization，ASLR)等保护技术的部署，实现代码注入攻击的难度越来越大，于是聪明的黑客又发明了 ROP 攻击及其各种变种，其主要思想是通过精确控制进程的执行流程，重新组合和复用可执行文件中已经存在的代码，实现恶意目的并绕过特定的防护技术和系统。目前针对 ROP 攻击较为有效的防护技术有不定期 ASLR 与控制流完整性约束(Control Flow Integrity，CFI)技术。粗粒度 ASLR 已经被确认不安全，而细粒度 ASLR 会导致代码膨胀从而增加了潜在的 Gadgets，并且很可能会使得库函数无法共享，从而严重影响了细粒度 ASLR 的实用性。

ROP 攻击首先要利用特定漏洞来实现栈上内容的覆盖，通过覆盖栈上的函数返回地址来实现控制流的修改，重新组合已有的代码并构造 Gadgets 链来实现恶意目的。ROP Gadgets 是指较短的汇编指令序列，一般以 ret 指令或其他间接跳转指令(例如 jmp eax 或 call eax 等)结束，每个 Gadget 仅仅实现功能非常有限的运算，但大量的 Gadgets 链接起来却可以实现任意功能。ROP 已被证明是图灵完备的。

下面的代码演示了在 PE 可执行文件中搜索 ROP Gadgets 的基本原理。这只是个基本的演示，并没有考虑更加复杂的情况。例如，如果控制流能够跳转到指令中间开始执行，就会打乱原有的指令序列而产生新的指令，从而产生原本不存在的 Gadget，这种情况这里没有考虑。您可以根据需要对下面的代码进行修改，或者阅读 ropper 工具的源代码以了解更多知识。

```
import time
import re
instructions = []
controlInstructions = ('call','ret','retn','jmp','jz','je','jnz','jne',
                       'js','jns','jo','jno','jp','jpe','jnp','jpo','jc',
                       'jb','jnae','jbe','jna','jnc','jnb','jae','jnbe',
                       'ja','jl','jnge','jnl','jge','jle','jng','jnle',
                       'jg','jcxz','jecxz')
def ReadInstructions():
```

```
    for seg_ea in Segments():
        for head in Heads(seg_ea,SegEnd(seg_ea)):
            if isCode(GetFlags(head)):
                #here using GetMnem(head) can get only mnemonic
                instruction =GetDisasm(head)
                instructions.append((hex(head),instruction))

    #print all the direct or indirect control instructions
    print 'The number of all instructions found is:',len(instructions)
    print 'And the direct or indirect control instructions are:'
    allControlInstructionsCount =0 #the number of control instructions

    #get all the mnemonics from instructions
    mnemonics = [t[1].split()[0] for t in instructions]

    for ins in controlInstructions:
        if ins in mnemonics:
            print ins,mnemonics.count(ins)
            allControlInstructionsCount =allControlInstructionsCount+mnemonics.count(ins)
    print 'The number of all control instructions is:',allControlInstructionsCount

#check if given instruction is a indirect control diversion instruction
def Check(instruction):
    if instruction.startswith('ret') or instruction.startswith('retn'):
        return True
    else:
        for instr in controlInstructions:
            if instr in ('ret', 'retn'):
                continue
            if instruction.startswith(instr +' e'):#like call edi
                return True
        return False

#output the potential gadgets
def Output(start, end):
    print '='*30
    for i in range(start, end+1):
        print instructions[i]

#find potential gadgets
def FindGadgets():
    total =len(instructions)
    gadgetNumber =0
    index =total -1
```

```
    while index >=0:
        instruction =instructions[index]
        if Check(instruction[1]):
            gadgetNumber +=1
            for i in range(1, 20):
                if Check(instructions[index - i][1]):
                    Output(index - i +1, index)
                    index =index - i
                    break
            else:
                Output(index - 19, index)
                index -=19
        else:
            index -=1
    print '=' * 30
    print 'Total number of gadgets:', gadgetNumber

start =time.time()
ReadInstructions()
FindGadgets()
print time.time() - start
```

16.2.2 Immunity Debugger 编程

Immunity Debugger 是一款使用 Python 开发的、非常成熟的调试器软件，支持软件调试的几乎所有功能，可以用于实现漏洞利用编写、模糊测试、恶意软件分析以及可执行文件的逆向工程分析，并且支持 PyCommand 接口以支持 Python 编程进行二次开发，其启动界面如图 16-4 所示。

在 Immunity Debugger 界面中的工具栏上单击左边第二个带有符号>>>的工具按钮，打开 Immunity Debugger Python Shell 窗口，在该窗口中可以直接执行 Python 语句，并通过对象 imm 来访问 Immlib 库的所有成员，如图 16-5 所示。或者，可以通过在 Immunity Debugger 主界面下面的命令框中输入“!”符号来执行编写好的 Python 程序完成分析任务，如图 16-6 所示。

编写自己的插件时，可以使用 IDLE 或记事本等任意文本编辑器编写 Python 源程序，然后将程序文件存放至 Immunity Debugger 安装目录下的 PyCommands 目录中，最后通过在 Immunity Debugger 主界面下方命令框中输入“!”后加上程序文件名称即可执行，如图 16-6 所示。下面再通过几个示例来演示如何利用 Python 编程实现 PE 文件分析，您也可以参考 Immunity Debugger 安装目录下的 PyCommands 目录中的文件来了解更多分析技巧，或者根据自己的软件分析需求来编写相应的插件。

图 16-4 Immunity Debugger 启动界面

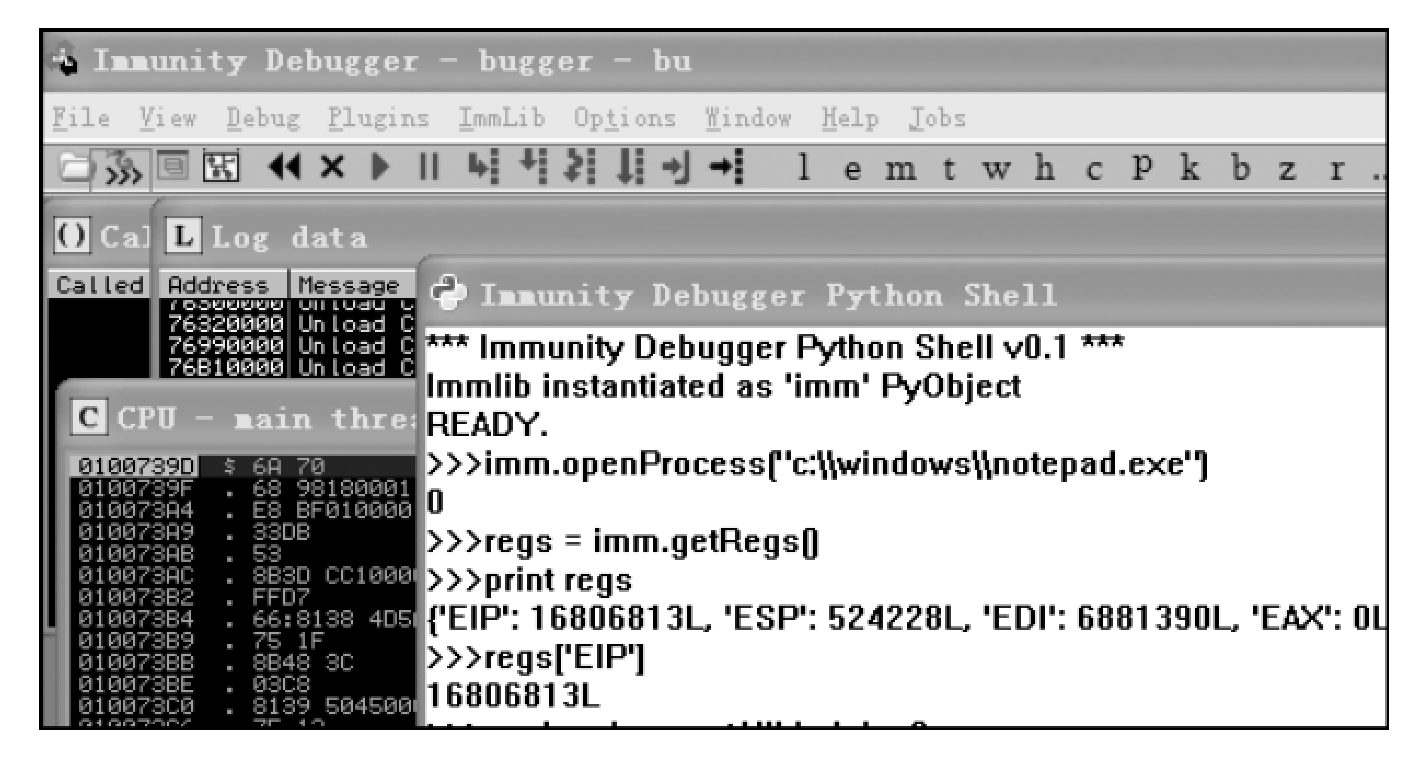


图 16-5 在 Immunity Debugger 中执行 Python 语句

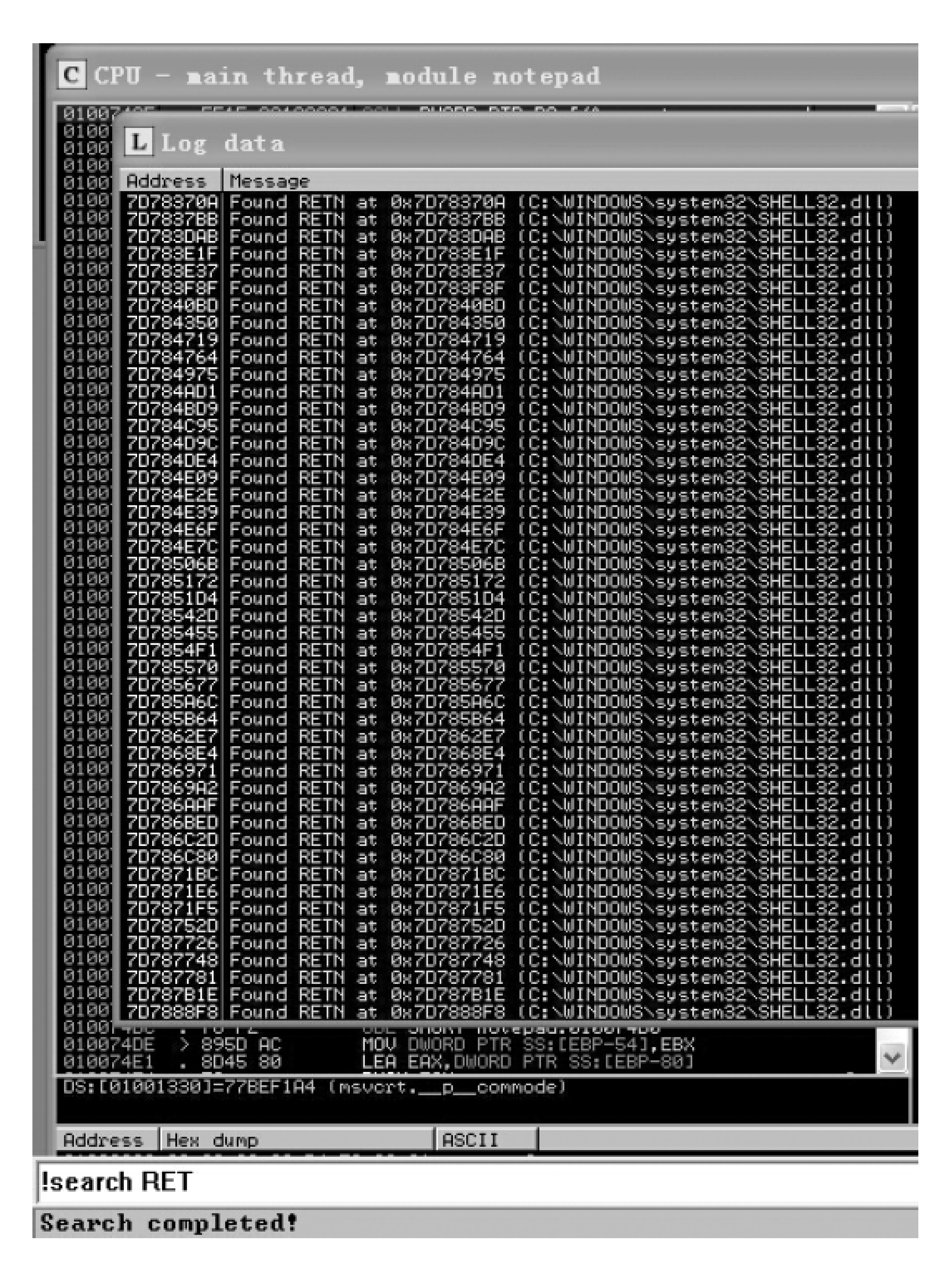


图 16-6　在 Immunity Debugger 中执行 Python 程序

1. 寻找可执行文件中的循环

```
from immlib import *
from immutils import *
import getopt

DESC=""" Find natural loops given a function start address """

def usage(imm):
    imm.log("!findloop -a <address>")
    imm.log("-a (function start address)")
    imm.log("-h This help")

def main(args):
    imm =Debugger()
    try:
        opts, argo =getopt.getopt(args, "a:")
    except:
        return usage(imm)
    for o,a in opts:
        if o =="-a":
            loops =imm.findLoops(int(a,16))
```

```
            for loop in loops:
                imm.log("LOOP! from:0x%08x, to:0x%08x"%(loop[0],loop[1]),loop[0])

                func = imm.getFunction(int(a,16))
                bbs = func.getBasicBlocks()

                #寻找第一个和最后一个节点
                first = 0xffffffff
                last = 0
                for node in loop[2]:
                    if node < first: first = node
                    if node > last: last = node

                #标记循环节点,但如果存在任何形式的注释就不做任何改变
                for node in loop[2]:
                    imm.log(" Loop node:0x%08x"%node,node)
                    for bb in bbs:
                        if bb.getStart() == node:
                            instrs = bb.getInstructions(imm)
                            for op in instrs:
                                if not imm.getComment(op.getAddress()) and op.getAddress
                                () != node:
                                    if node == last and op.getAddress() == instrs[-1].
                                    getAddress():
                                        #最后一个节点的最后一个指令
                                        imm.setComment(op.getAddress(), "/")
                                    else:
                                        imm.setComment(op.getAddress(), "|")

                    if not imm.getComment(node):
                        if node == first:
                            imm.setComment(node, "\ Loop 0x%08X Node"%(loop[0]))
                        else:
                            imm.setComment(node, "| Loop 0x%08X Node"%(loop[0]))

        return "Done!"
    if o == "-h":
        return usage(imm)
```

2. 寻找可执行文件中的打包器

```
import immlib
import getopt
import struct

DESC = """Find a Packer/Cryptor on a Module (Note: It might take some times due to the amount of
```

```
signature on our db)"""

def usage(imm):
    imm.log("!findpacker [-f] -m filename/module Get the RPC information of a loaded dll or
    for all loaded DLL's",focus=1)
    imm.log(" -m filename/module File or Module to search for")
    imm.log(" -f When set, it look in the file instead of the loaded module")
    imm.log(" ex: !findpacker -m notepad")
    imm.log("NOTE: It might take some times due to the amount of signature on our db")

def main(args):
    imm = immlib.Debugger()
    if not args:
        usage(imm)
        return "No args"
    try:
        opts, argo = getopt.getopt(args, "m:f")
    except getopt.GetoptError:
        usage(imm)
        return "Bad heap argument %s" %args[0]

    module = None
    OnMemory = 1

    for o,a in opts:
        if o == "-m":
            module = a
        elif o == '-f':
            OnMemory = 0

    if not module:
        usage(imm)
        return "No module provided, see the Log Window for details of usage"

    try:
        ret = imm.findPacker( module, OnMemory = OnMemory)
    except Exception, msg:
        return "Error: %s" %msg

    if not ret:
        return "No Packer found"

    for (addr, name) in ret:
        imm.log("Packer found!: %s at 0x%08x" % (name, addr), address = addr)
    return "Packers found on %s: %d" % (module, len(ret))
```

3. 寻找可执行文件中的指令

```
import immlib

DESC = "Search code in memory"

def usage(imm):
    imm.log("!searchcode Search code in memory")
    imm.log("!searchcode <asm code> ")

def main(args):
    imm = immlib.Debugger()

    look = " ".join(args)
    ret = imm.search( imm.assemble( look ) )

    for a in ret:

        module = imm.findModule(a)
        if not module:
            module = "none"
        else:
            module = module[0]

        # Grab the memory access type for this address
        page = imm.getMemoryPageByAddress( a )
        access = page.getAccess( human = True )

        imm.log("Found %s at 0x%08x [%s] Access: (%s)" %(look, a, module, access), address =
        a)
    if ret:
        return "Found %d address (Check the Log Window for details)" %len(ret)
    else:
        return "Sorry, no code found"
```

16.3 Windows 平台软件调试原理

不论使用什么语言开发软件调试器，其基本原理都是一致的，都是调用操作系统自身提供的调试接口，设置断点，并对被调试软件的执行过程以及有关事件进行跟踪和处理。在本节中，主要介绍 Windows 平台上的调试接口、调试事件和断点等基本概念与调试原理。

16.3.1 Windows 调试接口

在 Win32 中自带了大量支持不同类型应用开发的 API 函数，其中一部分被称为 Win32 调试 API(Win32 Debug API)，提供了编写软件调试器所需要的大部分功能。利用这些 API 可以

加载一个程序或将调试器捆绑到一个正在运行的进程上以供调试;可以获得被调试进程的深层信息,例如进程 ID、入口地址和映像基址等;甚至可以对被调试的程序进行任意的修改,包括进程的内存和线程的运行环境等。表 16-1 列出了常用的 Windows 调试 API。

表 16-1　常用的 Windows 调试 API

API 函数	功 能 说 明
ContinueDebugEvent()	恢复先前由于调试事件而挂起的线程
DebugActiveProcess()	将调试器捆绑到一个正在运行的进程上
DebugActiveProcessStop()	将调试器从一个正在运行的进程上卸载
DebugBreak()	在当前进程中产生一个断点异常,如果当前进程不是处在被调试状态,那么这个异常将被系统例程接管,多数情况下会导致当前进程被终止。与在程序中直接插入 INT 3 的效果一样
DebugBreakProcess()	在指定进程中产生一个断点异常
FatalExit()	将使调用进程强制退出,将控制权转移至调试器,在退出前会先调用一个 INT 3 断点
FlushInstructionCache()	刷新指令高速缓存
GetThreadContext()	获取指定线程的执行环境
GetThreadSelectorEntry()	返回指定选择器和线程的描述符表的入口地址
IsDebuggerPresent()	判断调用进程是否处于被调试环境中
OutputDebugString()	将一个字符串传递给调试器显示
ReadProcessMemory()	读取指定进程的某区域内的数据
SetThreadContext()	设置指定线程的执行环境
WaitForDebugEvent()	等待被调试进程发生调试事件
WriteProcessMemory()	在指定进程的某区域内写入数据

16.3.2　调试事件

调试器的主要工作是监视目标进程的执行并对目标进程执行过程中发生的每一个调试事件进行相应的响应和处理。当目标进程发生一个调试事件后,系统将会通知调试器来处理这个事件,调试器利用 WaitForDebugEvent()函数来获取目标进程中发生的调试事件信息。常用的调试事件如表 16-2 所示。

表 16-2　调试事件

调 试 事 件	含　　义
CREATE_PROCESS_DEBUG_EVENT	进程被创建。当调试的进程刚被创建(还未运行)或调试器开始调试已经激活的进程时,就会生成这个事件
CREATE_THEAD_DEBUG_EVENT	在调试进程中创建一个新的进程或调试器开始调试已经激活的进程时,就会生成这个调试事件。要注意的是,当调试的主线程被创建时不会收到该通知

续表

调试事件	含义
EXCEPTION_DEBUG_EVENT	在调试的进程中出现了异常，就会生成该调试事件
EXIT_PROCESS_DEBUG_EVENT	每当退出调试进程中的最后一个线程时，产生这个事件
EXIT_THREAD_DEBUG_EVENT	调试中的线程退出时事件发生，调试的主线程退出时不会收到该通知
LOAD_DLL_DEBUG_EVENT	每当被调试的进程装载 DLL 文件时，就生成这个事件。当 PE 装载器第一次解析出与 DLL 文件有关的链接时，将收到这一事件。调试进程使用了 LoadLibrary 时也会发生。每当 DLL 文件装载到地址空间中去时，都要调用该调试事件
OUTPUT_DEBUG_STRING_EVENT	当调试进程调用 DebugOutputString 函数向程序发送消息字符串时该事件发生
UNLOAD_DLL_DEBUG_EVENT	每当调试进程使用 FreeLibrary 函数卸载 DLL 文件时，就会生成该调试事件。仅当最后一次从过程的地址空间卸载 DLL 文件时，才出现该调试事件(也就是说 DLL 文件的使用次数为 0 时)
RIP_EVENT	只有 Windows 98 检查过的构件才会生成该调试事件。该调试事件是报告错误信息

当 WaitForDebugEvent()接收到一个调试事件时，会把调试事件的信息填写入 DEBUG_EVENT 结构中返回，然后检查 dwDebugEventCode 字段中的值，根据它来判断被调试的进程中发生了哪种类型的调试事件。在调试事件结构体中，dwProcessId 的值是调试事件所发生的进程的标识符，dwThreadId 的值是调试事件所发生的线程的标识符，最后一个是与调试事件类型 dwDebugEventCode 对应的共用体成员。

```
typedef struct _DEBUG_EVENT {
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
        RIP_INFO RipInfo;
    } u;
} DEBUG_EVENT;
```

16.3.3 进程调试

实际应用时，根据不同的需要，可以创建一个新的进程进行调试，也可以调试一个正在

运行的进程，不管哪种方式，都需要建立循环不断地接收和处理调试事件，进行相应的处理，然后等待下一个调试事件的触发。

1. 创建一个新进程以供调试

通过 CreateProcess()创建新进程时，如果在 dwCreationFlags 标志字段中设置了 DEBUG_PROCESS 或 DEBUG_ONLY_THIS_PROCESS 标志，将创建一个用以调试的新进程。

2. 调试一个已有进程

利用 DebugActiveProcess()函数可以将调试器捆绑到一个正在运行的进程上，如果执行成功，则效果类似于利用 DEBUG_ONLY_THIS_PROCESS 标志创建的新进程。要注意的是，在 NT 内核下当试图通过 DebugActiveProcess()函数将调试器捆绑到一个创建时带有安全描述符的进程上时，将被拒绝。

3. 建立事件监视循环

使用 WaitForDebugEvent()和 ContinueDebugEvent()函数建立循环来不断地监视调试事件。WaitForDebugEvent()在一段时间内等待目标进程中调试事件的发生，如果在这段时间没有调试事件发生，那么函数将返回 False；如果在指定时间内调试事件发生了，那么函数将返回 True，并且把所发生的调试事件及其相关信息填写入一个 DEBUG_EVENT 结构。然后调试器会检查这些信息，并据此进行相应的处理。在对这些事件进行相应的操作后，就可以使用 ContinueDebugEvent()函数来恢复线程的执行，并等待下一个调试事件的发生。需要注意的是，WaitForDebugEvent()只能使用在创建以供调试的或是已被捆绑调试器的进程中的某个线程上。下面的代码以 C 语言形式演示了该循环的构建方式。

```
PROCESS_INFORMATION pi;
STARTUP_INFO si;
DEBUG_EVENT devent;
if(CreateProcess(0,"target.exe",0,0,FALSE,DEBUG_ONLY_THIS_PROCESS,0,0,&si,pi))
{
    while(TRUE)
    {
        if(WaitForDebugEvent(&devent,100))    //在 100ms 内等待调试事件
        {
            switch (devent.dwDebugEventCode)
            {
                case CREATE_PROCESS_DEBUG_EVENT:
                    //在此处编写自己的处理代码
                    break;
                case EXIT_PROCESS_DEBUG_EVENT:
                    //在此处编写自己的处理代码
                    break;
                case EXCEPTION_DEBUG_EVENT:
                    //在此处编写自己的处理代码
                    break;
                //其他类型调试事件处理代码
```

```
                }
                ContinueDebugEvent(devent.dwProcessId, devent.dwThreadId,DBG_CONTINUE);
            }
            else
            {
              //其他一些操作
            }
        }
    }
    else
    {
        MessageBox(0,"Unexpected load error","Fatal Error" ,MB_OK);
    }
```

16.3.4 线程环境

每个进程都有一个最初的主线程,通过主线程可以创建在同一地址空间中运行的其他线程。进程并不执行代码,真正执行代码的是线程。同一个进程中的所有线程共享相同的地址空间和相同的系统资源,但是每个线程又有不同的执行环境。

Windows 分配给每个线程一个很短的时间片,时间片用完之后,系统将暂停当前线程并切换到下一个具有最高优先级的待调度线程。在切换之前,系统会把当前线程执行状态保存到一个名为 CONTEXT 的结构体中,包括线程执行所用寄存器、系统堆栈和用户堆栈、线程所用的描述符表等其他状态信息。当该线程再次被调度进入 CPU 运行时,系统将恢复上次保存的上下文,以便线程可以继续上一次未完成的工作。

在调试时,为了满足某些特定调试目的,也可以根据需要来读取和修改线程环境,具体步骤有 4 个。

(1) 调用 SuspendThread()函数暂停线程。

(2) 调用 GetThreadContext()函数读取线程环境。

(3) 修改读取到的数据,再调用 SetThreadContext()函数设置线程新的执行环境。

(4) 调用 ResumeThread()函数恢复线程执行。

16.3.5 断点

断点是最常用的软件调试技术之一,其基本思想是在某一个位置设置一个"陷阱",当 CPU 执行到这个位置时停止被调试的程序并中断到调试器中,让调试者进行分析和调试,调试者分析结束后,可以让被调试程序恢复执行。通过设置断点可以暂停程序执行,并可以观察和记录指令信息、变量值、堆栈参数和内存数据,还可以深入了解和把握程序执行的内部原理和详细过程,断点对于软件调试具有重要的意义和作用。

断点可以分为软件断点、硬件断点和内存断点三大类,也有的分为代码断点、数据断点和 I/O 断点三类,这里只介绍前一种分类标准。

1. 软件断点

软件断点是一个单字节指令(INT 3,字节码为 0xCC),可以在程序中设置多个软件断

点，使得程序执行到该处时能够暂停执行，并将控制权转移给调试器的断点处理函数。

当调试器被告知在目标地址设置一个断点，它首先读取目标地址的第一个字节的操作码，然后保存起来，同时把地址存储在内部的中断列表中。接着，调试器把一个字节操作码 0xCC 写入刚才的地址。当 CPU 执行到 0xCC 操作码时就会触发一个 INT 3 中断事件，此时调试器就能捕捉到这个事件。调试器继续判断这个发生中断事件的地址（通过指令指针寄存器 EIP）是不是自己先前设置断点的地址。如果在调试器内部的断点列表中找到了这个地址，就将设置断点前存储起来的操作码写回到目标地址，这样进程被调试器恢复后就能正常执行。

2. 硬件断点

硬件断点通过调试寄存器实现，设置在 CPU 级别上，当需要调试某个指定区域而又无法修改该区域时，硬件断点非常有用。

一个 CPU 一般会有 8 个调试寄存器（DR0～DR7），用于管理硬件断点。其中调试寄存器 DR0 到调试寄存器 DR3 存储硬件断点地址，同一时间内最多只能设置 4 个硬件断点；DR4 和 DR5 保留，DR6 是状态寄存器，说明被断点触发的调试事件的类型；DR7 本质上是一个硬件断点的开关寄存器，同时也存储了断点的不同类型。通过在 DR7 寄存器里设置不同标志，能够创建以下几种断点：当特定的地址上有指令执行时中断、当特定的地址上有数据写入时、当特定的地址上有数据读或者写但不执行时。

硬件断点使用 INT 1 实现，该中断负责硬件中断和步进事件。步进是指根据预定的流程一条一条地执行指令，每执行完一条指令后暂停下来，从而可以精确地观察关键代码并监视寄存器和内存数据的变化。在 CPU 每次执行代码之前，都会先确认当前将要执行代码的地址是否是硬件断点的地址，同时也要确认是否有代码要访问被设置了硬件断点的内存区域。如果任何储存在 DR0～DR3 中的地址所指向的区域被访问了，就会触发 INT 1 中断，同时暂停 CPU；如果不是中断地址则 CPU 执行该行代码，到下一行代码时，CPU 继续重复上面的过程。

3. 内存断点

内存断点是通过修改内存中指定块或页的访问权限来实现的。通过将指定内存块或页的访问权限属性设置为受保护的，则任何不符合访问权限约束的操作都将失败，并抛出异常，导致 CPU 暂停执行，使得调试器可以查看当前执行状态。

一般来说，每个内存块或页的访问权限都由 3 种不同的访问权限组成：是否可执行、是否可读、是否可写。每个操作系统都提供了用来查询和修改内存页访问权限的函数，在 Windows 操作系统中可以使用 VirtualProtect()函数来修改主调进程虚拟地址空间中已提交页面的保护属性，使用 VirtualProtectEx()函数可以修改其他进程虚拟地址空间页面的保护属性。

16.4 案例精选

本节中通过一些示例来演示如何使用 Python 编写程序分析 PE 文件，其中用到了不同的扩展库，请根据需要进行下载安装。

1. 利用 pefile 模块查看 PE 文件详细信息

```
>>> import pefile
>>> f = pefile.PE(r'C:\windows\notepad.exe')
>>> f
<pefile.PE instance at 0x0164CB98>
>>> print f
```

输出结果(略)

```
>>> print f.FILE_HEADER
[IMAGE_FILE_HEADER]
0xE4     0x0 Machine:                          0x14C
0xE6     0x2 NumberOfSections:                 0x3
0xE8     0x4 TimeDateStamp:                    0x48025287 [Sun Apr 13 18:35:51 2008 UTC]
0xEC     0x8 PointerToSymbolTable:             0x0
0xF0     0xC NumberOfSymbols:                  0x0
0xF4     0x10 SizeOfOptionalHeader:            0xE0
0xF6     0x12 Characteristics:                 0x10F
>>> print f.OPTIONAL_HEADER
[IMAGE_OPTIONAL_HEADER]
0xF8     0x0 Magic:                            0x10B
0xFA     0x2 MajorLinkerVersion:               0x7
0xFB     0x3 MinorLinkerVersion:               0xA
0xFC     0x4 SizeOfCode:                       0x7800
0x100    0x8 SizeOfInitializedData:            0x8800
0x104    0xC SizeOfUninitializedData:          0x0
0x108    0x10 AddressOfEntryPoint:             0x739D
0x10C    0x14 BaseOfCode:                      0x1000
0x110    0x18 BaseOfData:                      0x9000
0x114    0x1C ImageBase:                       0x1000000
0x118    0x20 SectionAlignment:                0x1000
0x11C    0x24 FileAlignment:                   0x200
0x120    0x28 MajorOperatingSystemVersion:     0x5
0x122    0x2A MinorOperatingSystemVersion:     0x1
0x124    0x2C MajorImageVersion:               0x5
0x126    0x2E MinorImageVersion:               0x1
0x128    0x30 MajorSubsystemVersion:           0x4
0x12A    0x32 MinorSubsystemVersion:           0x0
0x12C    0x34 Reserved1:                       0x0
0x130    0x38 SizeOfImage:                     0x12F20
0x134    0x3C SizeOfHeaders:                   0x400
0x138    0x40 CheckSum:                        0x18ADA
0x13C    0x44 Subsystem:                       0x2
0x13E    0x46 DllCharacteristics:              0x8000
0x140    0x48 SizeOfStackReserve:              0x40000
```

```
0x144     0x4C SizeOfStackCommit:               0x11000
0x148     0x50 SizeOfHeapReserve:               0x100000
0x14C     0x54 SizeOfHeapCommit:                0x1000
0x150     0x58 LoaderFlags:                     0x0
0x154     0x5C NumberOfRvaAndSizes:             0x10
>>> for k in f.sections:
    print k
```

输出结果(略)

```
>>> f.is_dll()
False
>>> f.is_exe()
True
```

2. 利用 pefile 模块枚举 DLL 的导出项

```
>>> import pefile
>>> pe = pefile.PE(r'C:\windows\glut32.dll')
>>> if hasattr(pe, 'DIRECTORY_ENTRY_EXPORT'):
    for exp in pe.DIRECTORY_ENTRY_EXPORT.symbols:
        print hex(pe.OPTIONAL_HEADER.ImageBase+exp.address),\
            exp.name, exp.ordinal
```

输出结果(略)

3. 利用 pefile 和 pydasm 模块从 PE 文件入口点开始反汇编

```
import pefile
import pydasm
import sys

pe = pefile.PE(r"C:\windows\notepad.exe")

console = sys.stdout
f = open('pe_dasm.txt', 'w')
sys.stdout = f

ep = pe.OPTIONAL_HEADER.AddressOfEntryPoint
ep_ava = ep + pe.OPTIONAL_HEADER.ImageBase
data = pe.get_memory_mapped_image()[ep:]
offset = 0
while offset < len(data):
    i = pydasm.get_instruction(data[offset:], pydasm.MODE_32)
    instruction = pydasm.get_instruction_string(i, pydasm.FORMAT_INTEL, ep_ava+offset)
    if instruction != None:
        print hex(ep + offset), '\t', instruction
    else:
        break
```

```
    try:
        offset += i.length
    except BaseException,e:
        break

f.close()
sys.stdout = console
```

4. 利用 WinAppDbg 监视 Windows API 调用

```
from winappdbg import Debug, EventHandler
import sys
import os

class MyEventHandler( EventHandler ):
    # Add the APIs you want to hook
    apiHooks = {'kernel32.dll' : [( 'CreateFileW', 7)]}

    # The pre_ functions are called upon entering the API
    def pre_CreateFileW(self, event, ra, lpFileName, dwDesiredAccess,
            dwShareMode, lpSecurityAttributes, dwCreationDisposition,
                        dwFlagsAndAttributes, hTemplateFile):
        fname = event.get_process().peek_string(lpFileName, fUnicode=True)
        print "CreateFileW: %s" %(fname)

    # The post_ functions are called upon exiting the API
    def post_CreateFileW(self, event, retval):
        if retval:
            print 'Suceeded (handle value: %x)' %(retval)
        else:
            print 'Failed!'

if __name__ == "__main__":
    if len(sys.argv) < 2 or not os.path.isfile(sys.argv[1]):
        print "\nUsage: %s <File to monitor> [arg1, arg2, ...]\n" %sys.argv[0]
        sys.exit()

    # Instance a Debug object, passing it the MyEventHandler instance
    debug = Debug( MyEventHandler() )
    try:
        # Start a new process for debugging
        p = debug.execv(sys.argv[1:], bFollow=True)
        # Wait for the debugged process to finish
        debug.loop()
# Stop the debugger
finally:
```

```
debug.stop()
```

将上面的代码保存为 simpleapi.py，使用方法与运行结果如图 16-7 所示。

```
C:\Python27>python simpleapi.py c:\windows\notepad.exe a.txt
CreateFileW: C:\WINDOWS\WindowsShell.Manifest
Suceeded (handle value: 48)
CreateFileW: C:\WINDOWS\system32\msctfime.ime
Suceeded (handle value: 9c)
CreateFileW: C:\WINDOWS\system32\msctfime.ime
Suceeded (handle value: 9c)
CreateFileW: a.txt
Suceeded (handle value: ac)
CreateFileW: C:\Documents and Settings\Administrator\Application Data\TaobaoProt
ect\cg.dat
Suceeded (handle value: cc)

C:\Python27>
```

图 16-7　Windows API 监视器

本章知识精要

(1) 在 Windows 平台上，exe 文件、com 文件、dll 文件、ocx 文件、sys 文件和 scr 文件等都属于 PE 文件。

(2) PE 文件规范最新版本是 2013 年 2 月 6 日发布的 8.3 版。

(3) 在分析软件尤其是恶意软件时，应尽量使用虚拟机或沙箱系统，避免本地物理主机系统被感染而造成不必要的损失。

(4) IDA、W32DASM 是成熟的可执行文件反汇编工具，OllyDbg、WinDbg 和 Immunity Debugger 是成熟的软件调试工具。

(5) 通过 IDAPython 插件可以在 IDA 中运行 Python 程序实现自定义的软件测试与分析功能。

(6) ROP、JOP 是近几年流行的攻击方式，目前比较有效的防范技术是 CFI。

(7) 软件调试时经常需要设置断点，常见的断点类型有软件断点、硬件断点和内存断点。

(8) 调试器的主要工作就是监视目标进程的运行并对目标执行过程中发生的每一个调试事件进行相应的反应和处理。

习　题

1. 下载 PE 文件规范 8.3 版本，并尝试了解 PE 文件的基本结构。
2. 下载并安装 IDA Pro 与 Immunity Debugger，并简单了解 PE 文件反汇编和调试步骤。
3. 安装并配置 IDAPython 插件，然后运行本章 16.2.1 节的 Python 代码。
4. 在 Immunity Debugger 调试器中运行本章 16.2.2 节中的代码。
5. 叙述软件调试断点的概念、作用及其分类。
6. 运行 16.4 节中的代码并查看运行结果。

第 17 章　科学计算与可视化

用于科学计算与可视化的 Python 模块非常多，例如 NumPy、SciPy、SymPy、Matplotlib、Traits、TraitsUI、Chaco、TVTK、Mayavi、VPython 和 OpenCV。其中，NumPy 模块是科学计算包，提供了 Python 中没有的数组对象，支持 N 维数组运算、处理大型矩阵、成熟的广播函数库、矢量运算、线性代数、傅里叶变换以及随机数生成等功能，并可与 C++ 、FORTRAN 等语言无缝结合。SciPy 模块依赖于 NumPy，提供了更多的数学工具，包括矩阵运算、线性方程组求解、积分和优化等。Matplotlib 是比较常用的绘图模块，可以快速地将计算结果以不同类型的图形展示出来。如果您需要了解更多科学计算与可视化模块，可以参考网页 http://www.lfd.uci.edu/～gohlke/pythonlibs/。

在本章中，主要通过大量的示例代码来介绍 NumPy、SciPy、Matplotlib 几个模块的应用，如果您的计算机上还没有安装，请自行下载并安装所需要的模块。

17.1　NumPy 简单应用

根据大多数人的习惯，往往会使用下面的方式来导入 NumPy 模块：

```
>>> import numpy as np
```

1. 生成数组

```
>>> a = np.array((1,2,3,4,5))
>>> b = np.array(([1,2,3],[4,5,6],[7,8,9]))
>>> x = np.linspace(0,5,10)
>>> x
array([ 0.        , 0.55555556, 1.11111111, 1.66666667, 2.22222222,
        2.77777778, 3.33333333, 3.88888889, 4.44444444, 5.        ])
>>> y = np.logspace(0,100,10)
>>> y
array([ 1.00000000e+000, 1.29154967e+011, 1.66810054e+022,
         2.15443469e+033, 2.78255940e+044, 3.59381366e+055,
         4.64158883e+066, 5.99484250e+077, 7.74263683e+088,
         1.00000000e+100])
```

2. 数组与数值的算术运算

```
>>> a = np.array((1,2,3,4,5))
>>> a * 2
array([ 2, 4, 6, 8, 10])
>>> a/2
array([0, 1, 1, 2, 2])
```

```
>>>a/2.0
array([ 0.5, 1. , 1.5, 2. , 2.5])
>>>a* * 2
array([ 1, 4, 9, 16, 25])
```

3. 数组与数组的算术运算

```
>>>a =np.array((1,2,3))
>>>b =np.array(([1,2,3],[4,5,6],[7,8,9]))
>>>c =a*b
>>>print c
[[ 1 4 9]
 [ 4 10 18]
 [ 7 16 27]]
>>>print c/b
[[1 2 3]
 [1 2 3]
 [1 2 3]]
>>>a =np.array((1,2,3))
>>>b =np.array((1,2,3))
>>>a*b
array([1, 4, 9])
>>>a+b
array([2, 4, 6])
```

4. 二维数组转置

```
>>>b =np.array(([1,2,3],[4,5,6],[7,8,9]))
>>>print b
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>>print b.T
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

5. 向量点积

```
>>>import numpy as np
>>>a=np.array((5,6,7))
>>>b=np.array((6,6,6))
>>>print np.dot(a,b)
108
```

6. 数组元素访问

```
>>>import numpy as np
>>>b =np.array(([1,2,3],[4,5,6],[7,8,9]))
```

```
>>>print b
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>>b[0,0]
1
>>>b[0][2]
3
```

数组元素还支持多元素同时访问，例如：

```
>>>x =np.arange(0,100,10,dtype=np.floating)
>>>x
array([ 0., 10., 20., 30., 40., 50., 60., 70., 80., 90.])
>>>index =np.random.randint(0,len(x),5)
>>>index
array([9, 6, 3, 9, 7])
>>>noise =np.random.standard_normal(5) * 0.3
>>>noise
array([ 0.43460475, 0.57262955, -0.15114837, 0.02738525, -0.01063617])
>>>x[index]
array([ 90., 60., 30., 90., 70.])
>>>x[index] +=noise
>>>x[index]
array([ 90.02738525, 60.57262955, 29.84885163, 90.02738525, 69.98936383])
>>>x
array([ 0.        , 10.        , 20.        , 29.84885163,
        40.        , 50.        , 60.57262955, 69.98936383,
        80.        , 90.02738525])
>>>x[1]
10.0
>>>x[[1,3,5]]
array([ 10.        , 29.84885163, 50.        ])
```

7. 三角函数运算

```
>>>b =np.array(([1,2,3],[4,5,6],[7,8,9]))
>>>print np.sin(b)
[[ 0.84147098 0.90929743 0.14112001]
 [-0.7568025 -0.95892427 -0.2794155 ]
 [ 0.6569866 0.98935825 0.41211849]]
```

8. 四舍五入

```
>>>print np.round(np.sin(b))
[[ 1. 1. 0.]
 [-1. -1. 0.]
 [ 1. 1. 0.]]
```

9. 对矩阵不同维度上的元素进行求和

```
>>> x = np.arange(0,10).reshape(2,5)
>>> x
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> np.sum(x)
45
>>> np.sum(x, axis=0)
array([ 5, 7, 9, 11, 13])
>>> np.sum(x, axis=1)
array([10, 35])
```

10. 计算矩阵不同维度上元素的均值

```
>>> x = np.arange(0,10).reshape(2,5)
>>> np.average(x, axis = 0)
array([ 2.5, 3.5, 4.5, 5.5, 6.5])
>>> np. average(x,axis=1)
array([ 2., 7.])
```

11. 计算数据的标准差与方差

```
>>> x = np.random.randint(0,10,size= (3,3))
>>> x
array([[4, 2, 8],
       [0, 8, 9],
       [0, 2, 7]])
>>> np.std(x)
3.4029761846919007
>>> np.std(x,axis=1)
array([ 2.49443826, 4.02768199, 2.94392029])
>>> np.var(x)
11.580246913580245
```

12. 对矩阵不同维度上的元素求最大值

```
>>> x
array([[4, 2, 8],
       [0, 8, 9],
       [0, 2, 7]])
>>> np.max(x)
9
>>> np.max(x,axis=1)
array([8, 9, 7])
```

13. 对矩阵不同维度上的元素进行排序

```
>>> x
```

```
array([[4, 2, 8],
       [0, 8, 9],
       [0, 2, 7]])
>>>np.sort(x)
array([[2, 4, 8],
       [0, 8, 9],
       [0, 2, 7]])
>>>np.sort(x,axis=0)
array([[0, 2, 7],
       [0, 2, 8],
       [4, 8, 9]])
```

14. 生成特殊数组

```
>>>print np.zeros((3,3))
[[ 0. 0. 0.]
 [ 0. 0. 0.]
 [ 0. 0. 0.]]
>>>print np.ones((3,3))
[[ 1. 1. 1.]
 [ 1. 1. 1.]
 [ 1. 1. 1.]]
>>>print np.identity(3)
[[ 1. 0. 0.]
 [ 0. 1. 0.]
 [ 0. 0. 1.]]
>>>np.empty((3,3))             #只申请空间,不初始化,速度很快
array([[ 4.24510694e+175, 5.03061214e+223, 4.72100120e+164],
       [ 2.63551414e-144, -1.00000000e+000, 0.00000000e+000],
       [ 0.00000000e+000, 0.00000000e+000, 1.00000000e+000]])
```

15. 改变数组大小

```
>>>a =np.arange(1,11,1)
>>>a
array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
>>>a.shape =2,5
>>>a
array([[ 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10]])
>>>a.shape =5,-1               #-1 表示自动计算
>>>a
array([[ 1, 2],
       [ 3, 4],
       [ 5, 6],
       [ 7, 8],
       [ 9, 10]])
```

```
>>>b =a.reshape(2,5)
>>>b
array([[ 1, 2, 3, 4, 5],
       [ 6, 7, 8, 9, 10]])
```

16. 切片操作

```
>>>a =np.arange(10)
>>>a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>>a[::-1]
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
>>>a[::2]
array([0, 2, 4, 6, 8])
>>>a[:5]
array([0, 1, 2, 3, 4])
>>>c
array([[ 0, 1, 2, 3, 4, 5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
>>>c[0,3:5]
array([3, 4])
>>>c[0]
array([0, 1, 2, 3, 4, 5])
>>>c[2:5,2:5]
array([[22, 23, 24],
       [32, 33, 34],
       [42, 43, 44]])
```

17. 布尔运算

```
>>>x =np.random.rand(10)
>>>x
array([ 0.93874098, 0.97312716, 0.45264749, 0.74117525, 0.89758246,
        0.29755703, 0.2182093 , 0.5673035 , 0.90745768, 0.71920431])
>>>x>0.5
array([ True, True, False, True, True, False, False, True, True, True], dtype=bool)
>>>x[x>0.5]
array([ 0.93874098, 0.97312716, 0.74117525, 0.89758246, 0.5673035 ,
        0.90745768, 0.71920431])
>>>np.array([1,2,3]) <np.array([3,2,1])
array([ True, False, False], dtype=bool)
>>>np.array([1,2,3]) ==np.array([3,2,1])
array([False, True, False], dtype=bool)
```

18. 取整运算

```
>>> x = np.random.rand(10) * 50
>>> x
array([ 0.69708323, 14.99931488, 15.04431214, 24.60547929,
        12.12020273, 42.72638176, 16.01128916, 38.91558471,
        39.6877989 , 21.98678429])
>>> np.array([t-int(t) for t in x])
array([ 0.69708323, 0.99931488, 0.04431214, 0.60547929, 0.12020273,
        0.72638176, 0.01128916, 0.91558471, 0.6877989 , 0.98678429])
```

19. 广播

```
>>> a = np.arange(0,60,10).reshape(-1,1)
>>> b = np.arange(0,6)
>>> a
array([[ 0],
       [10],
       [20],
       [30],
       [40],
       [50]])
>>> b
array([0, 1, 2, 3, 4, 5])
>>> a+b
array([[ 0, 1, 2, 3, 4, 5],
       [10, 11, 12, 13, 14, 15],
       [20, 21, 22, 23, 24, 25],
       [30, 31, 32, 33, 34, 35],
       [40, 41, 42, 43, 44, 45],
       [50, 51, 52, 53, 54, 55]])
```

20. 分段函数

```
>>> x = np.random.randint(0,10,size=(1,10))
>>> x
array([[0, 4, 3, 3, 8, 4, 7, 3, 1, 7]])
>>> np.where(x<5,0,1)
array([[0, 0, 0, 0, 1, 0, 1, 0, 0, 1]])
>>> x = np.random.randint(0,10,size=(1,10))
>>> x
array([[3, 6, 5, 1, 0, 7, 3, 9, 6, 0]])
>>> np.piecewise(x,[x>7,x<4],[lambda x:x*2,lambda x:x*3,0])
array([[ 9, 0, 0, 3, 0, 0, 9, 18, 0, 0]])
```

21. 计算唯一值以及出现次数

```
>>> x = np.random.randint(0,10,10)
```

```
>>>x
array([4, 7, 3, 6, 7, 4, 1, 9, 4, 8])
>>>np.bincount(x)
array([0, 1, 0, 1, 3, 0, 1, 2, 1, 1])
>>>np.unique(x)
array([1, 3, 4, 6, 7, 8, 9])
```

22. 计算加权平均值

```
>>>x =np.random.randint(0,10,10)
>>>x
array([7, 8, 5, 8, 0, 7, 9, 9, 9, 7])
>>>y =np.array([round(i,1) for i in list(np.random.random(10))])
>>>y
array([ 0.6, 0.8, 0.8, 0. , 0.6, 0.1, 0. , 0.2, 0.8, 0.7])
>>>np.sum(x * y)/np.sum(np.bincount(x))
2.9199999999999999
```

23. 矩阵运算

```
>>>import numpy as np
>>>a_list =[3,5,7]
>>>a_mat =np.matrix(a_list)
>>>a_mat
matrix([[3, 5, 7]])
>>>np.shape(a_mat)
(1, 3)
>>>b_mat =np.matrix((1,2,3))
>>>b_mat
matrix([[1, 2, 3]])
>>>a_mat * b_mat.T
matrix([[34]])
>>>a_mat.argsort()      #返回每个元素的排序序号
matrix([[0, 1, 2]])
>>>a_mat.mean()
5.0
>>>a_mat.sum()
15
>>>a_mat.max()
7
```

17.2 SciPy 简单应用

SciPy 是在 NumPy 的基础上增加了大量用于数学计算、科学计算以及工程计算的模块，包括线性代数、常微分方程数值求解、信号处理、图像处理和稀疏矩阵等。SciPy 主要模块如表 17-1 所示。

表 17-1 SciPy 主要模块

模 块	说 明
constants	常数
special	特殊函数
optimize	数值优化算法，如最小二乘拟合（leastsq）、函数最小值（fmin 系列）、非线性方程组求解（fsolve）等
interpolate	插值（interp1d、interp2d 等）
integrate	数值积分
signal	信号处理
ndimage	图像处理，包括 filters 滤波器模块、fourier 傅里叶变换模块、interpolation 图像插值模块、measurements 图像测量模块、morphology 形态学图像处理模块等
stats	统计

17.2.1 常数与特殊函数

SciPy 的 constants 模块包含了大量用于科学计算的常数，详情可以查看 http://docs.scipy.org/doc/scipy/reference/constants.html。

例如，可以使用下面的方法来访问该模块中预定义的常数：

```
>>> from scipy import constants as C
>>> C.c                       #真空中的光速
299792458.0
>>> C.h                       #普朗克常数
6.62606896e-34
>>> C.mile                    #一英里等于多少米
1609.3439999999998
>>> C.inch                    #一英寸等于多少米
0.0254
>>> C.degree                  #一度等于多少弧度
0.017453292519943295
>>> C.minute                  #一分钟等于多少秒
60.0
```

此外，SciPy 模块的 special 模块包含大量函数库，包括基本数学函数、特殊函数以及 NumPy 中的所有函数。

```
>>> from scipy import special as S
>>> x = [0, np.pi/2, np.pi, np.pi * 1.5, np.pi * 2]
>>> S.sin(x)
array([ 0.00000000e+00, 1.00000000e+00, 1.22464680e-16,
       -1.00000000e+00, -2.44929360e-16])
>>> x = [1, 2+3j, 4-5j]
>>> S.conjugate(x)            #共轭复数
```

```
array([ 1.-0.j, 2.-3.j, 4.+5.j])
>>>S.gamma(4)              #gamma 函数
6.0
```

17.2.2 SciPy 简单应用

中值滤波是数字信号处理、数字图像处理中常用的预处理技术。该技术的特点是将信号中每个值都替换为其邻域内的中值，即邻域内所有值排序后中间位置上的值。下面通过两个示例来演示 SciPy 模块中中值滤波的实现和应用。

```
>>>import random
>>>import numpy as np
>>>import scipy.signal as signal
>>>x =np.arange(0,100,10)
>>>random.shuffle(x)
>>>x
array([40, 0, 60, 20, 50, 70, 80, 90, 30, 10])
>>>signal.medfilt(x,3)
array([ 0., 40., 20., 50., 50., 70., 80., 80., 30., 10.])
```

下面的代码使用中值滤波实现了信号去噪，并将处理前后的信号值进行了对比：

```
import numpy as np
import scipy.signal as signal
x =np.arange(0,6,0.1)
y =np.sin(x)
z =y.copy()
print '=' * 20
print 'y:'
print y
print '=' * 20
print 'before adding noise.z-y:'
print z-y
index =np.random.randint(0,len(x),20)
noise =np.random.standard_normal(20) * 0.8
z[index]+=noise
print '=' * 20
print 'after adding noise.z-y:'
print z-y
result =signal.medfilt(z,3)
print '=' * 20
print 'after median filtering.z-y:'
print result-y
```

运行结果如下，可以看到，经过中值滤波处理之后，信号整体更加接近原始值，但同时也导致了一些微小的失真。另外，由于使用了随机数，运行该程序时得到的结果与下面看到的

可能会略有不同。

```
====================
y:
[ 0.          0.09983342  0.19866933  0.29552021  0.38941834  0.47942554
  0.56464247  0.64421769  0.71735609  0.78332691  0.84147098  0.89120736
  0.93203909  0.96355819  0.98544973  0.99749499  0.9995736   0.99166481
  0.97384763  0.94630009  0.90929743  0.86320937  0.8084964   0.74570521
  0.67546318  0.59847214  0.51550137  0.42737988  0.33498815  0.23924933
  0.14112001  0.04158066 -0.05837414 -0.15774569 -0.2555411  -0.35078323
 -0.44252044 -0.52983614 -0.61185789 -0.68776616 -0.7568025  -0.81827711
 -0.87157577 -0.91616594 -0.95160207 -0.97753012 -0.993691   -0.99992326
 -0.99616461 -0.98245261 -0.95892427 -0.92581468 -0.88345466 -0.83226744
 -0.77276449 -0.70554033 -0.63126664 -0.55068554 -0.46460218 -0.37387666]
====================
before adding noise.z-y:
[ 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.
  0.  0.  0.  0.  0.  0.]
====================
after adding noise.z-y:
[ 0.25949491  0.          0.         -1.50601958  0.          0.
 -0.57822021  0.          1.69250421  0.          0.          0.          0.
  0.          0.         -0.06843014  1.51711964  0.         -0.92222652
  0.          0.          0.          0.48069333  0.          0.55413147
  0.         -0.63580858 -0.62972023  0.          0.          0.          0.
  0.          0.          0.          0.          0.          0.
 -0.61507477  0.          0.01639333  0.          0.          0.
 -1.10931122  0.          0.          0.          0.         -0.71545621
  0.          0.          0.         -0.39571393  0.          0.08390631
  0.          0.          0.          0.        ]
====================
after median filtering.z-y:
[ 0.09983342  0.09883591 -0.09883591 -0.09685088  0.         -0.0900072
 -0.08521693  0.          0.06597082  0.05814408  0.          0.          0.
  0.         -0.02189154 -0.01204526 -0.00790879  0.         -0.02754754
 -0.03700266  0.          0.04608806  0.05471296  0.48388944  0.07024203
  0.         -0.63580858 -0.54768709 -0.09573882  0.          0.          0.
  0.          0.          0.          0.          0.          0.
 -0.07590827 -0.05264301  0.01639333  0.          0.          0.
 -0.02592804 -0.01616089  0.          0.00375865 -0.00375865 -0.013712   0.
  0.         -0.04236003 -0.05118721  0.          0.07427369  0.00963262
  0.          0.          0.        ]
```

17.3 Matplotlib 简单应用

Matplotlib 模块依赖于 NumPy 模块和 tkinter 模块，它可以绘制多种形式的图形，包括线图、直方图、饼状图、散点图和误差线图等，是计算结果可视化的重要工具。

1. 绘制正弦曲线

```
>>>import numpy as np
>>>import pylab as pl
>>>t =np.arange(0.0, 2.0 * np.pi, 0.01)
>>>s =np.sin(t)
>>>pl.plot(t,s)
>>>pl.xlabel('x')
>>>pl.ylabel('y')
>>>pl.title('sin')
>>>pl.show()
```

运行结果如图 17-1 所示。

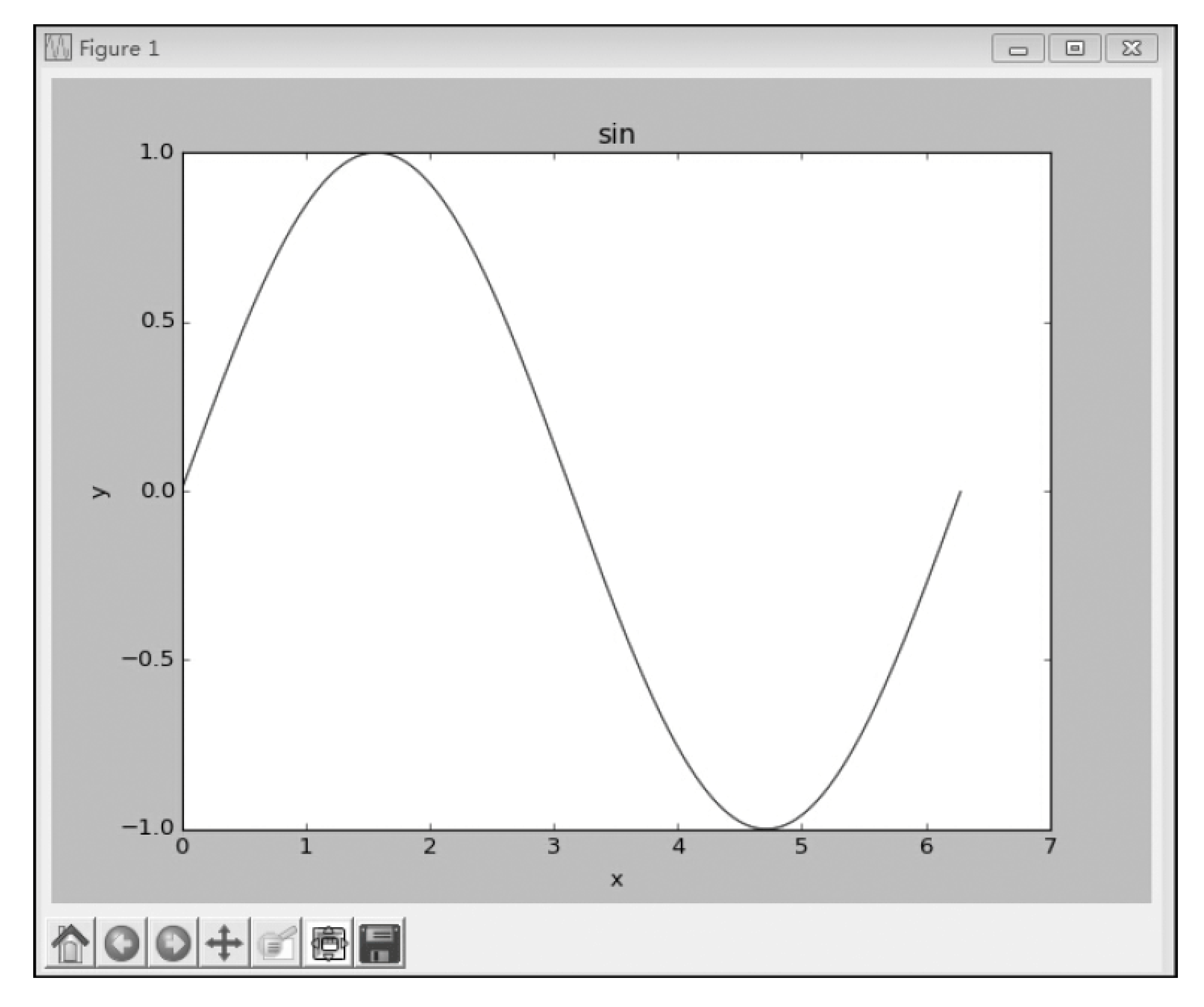


图 17-1 使用 Matplotlib 绘制正弦曲线

2. 绘制散点图

```
>>>a =np.arange(0, 2.0 * np.pi, 0.1)
>>>b =np.cos(a)
>>>pl.scatter(a,b)
>>>pl.show()
```

运行结果如图 17-2 所示。

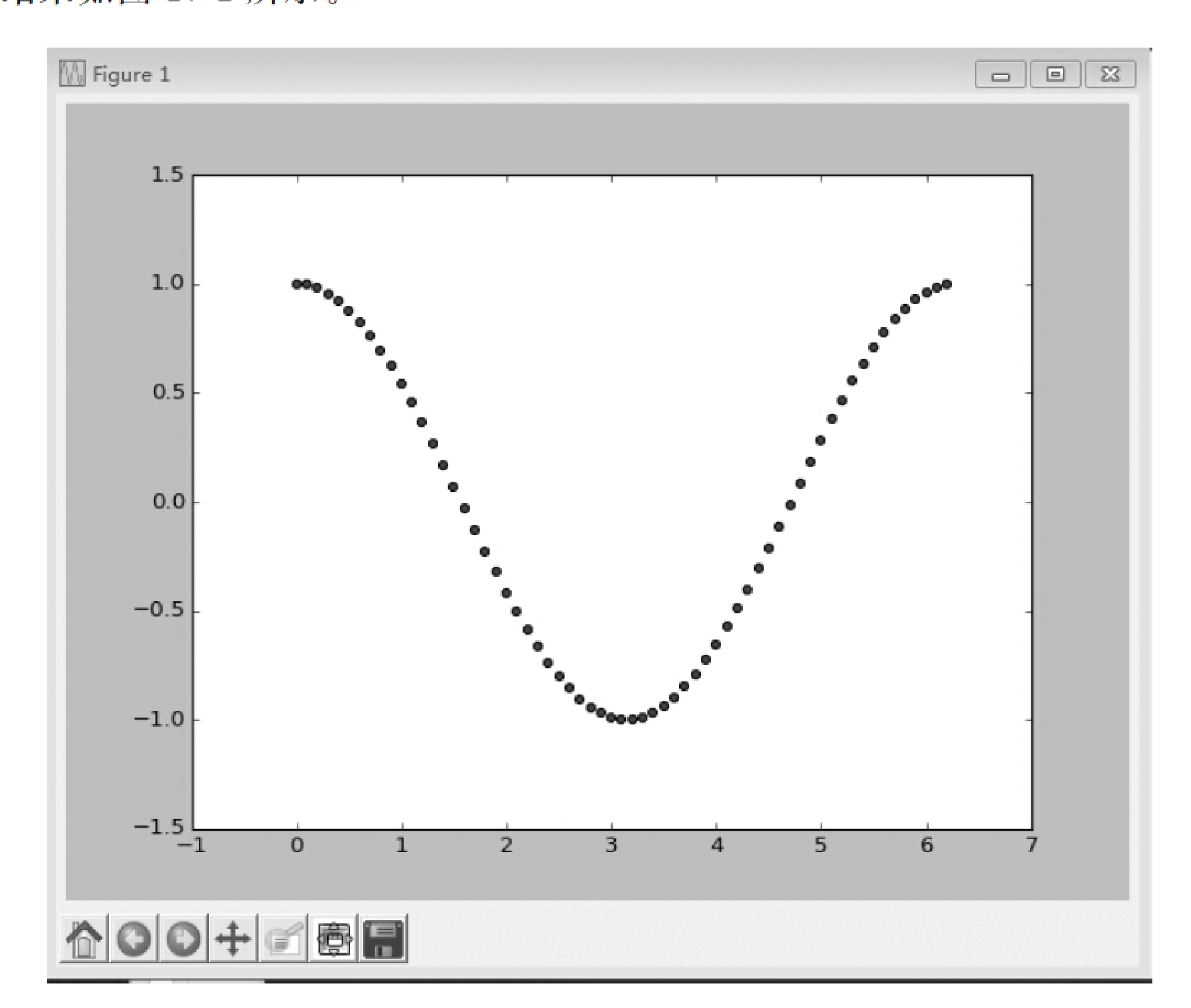


图 17-2 绘制余弦曲线散点图

下面的代码使用随机数生成数值,并根据数值大小来计算散点的大小。

```
>>>import matplotlib.pylab as pl
>>>import numpy as np
>>>x =np.random.random(100)
>>>y =np.random.random(100)
>>>pl.scatter(x,y,s=x * 500,c=u'r',marker=u' * ')       #s指大小,c指颜色,marker指符号形状
>>>pl.show()
```

运行结果如图 17-3 所示。

3. 使用 pyplot 绘制,多个图形在一起显示

```
import numpy as np
import matplotlib.pyplot as plt
x =np.linspace(0, 2 * np.pi, 500)
y =np.sin(x)
z =np.cos(x * x)
plt.figure(figsize= (8,4))
#标签前后加$将使用内嵌的 LaTex 引擎将其显示为公式
plt.plot(x,y,label= '%sin(x)$ ',color= 'red',linewidth=2)   #红色,2个像素宽
plt.plot(x,z,'b- - ',label= '$ cos(x^2)$ ')                 #蓝色,虚线
plt.xlabel('Time(s)')
plt.ylabel('Volt')
plt.title('Sin and Cos figure using pyplot')
```

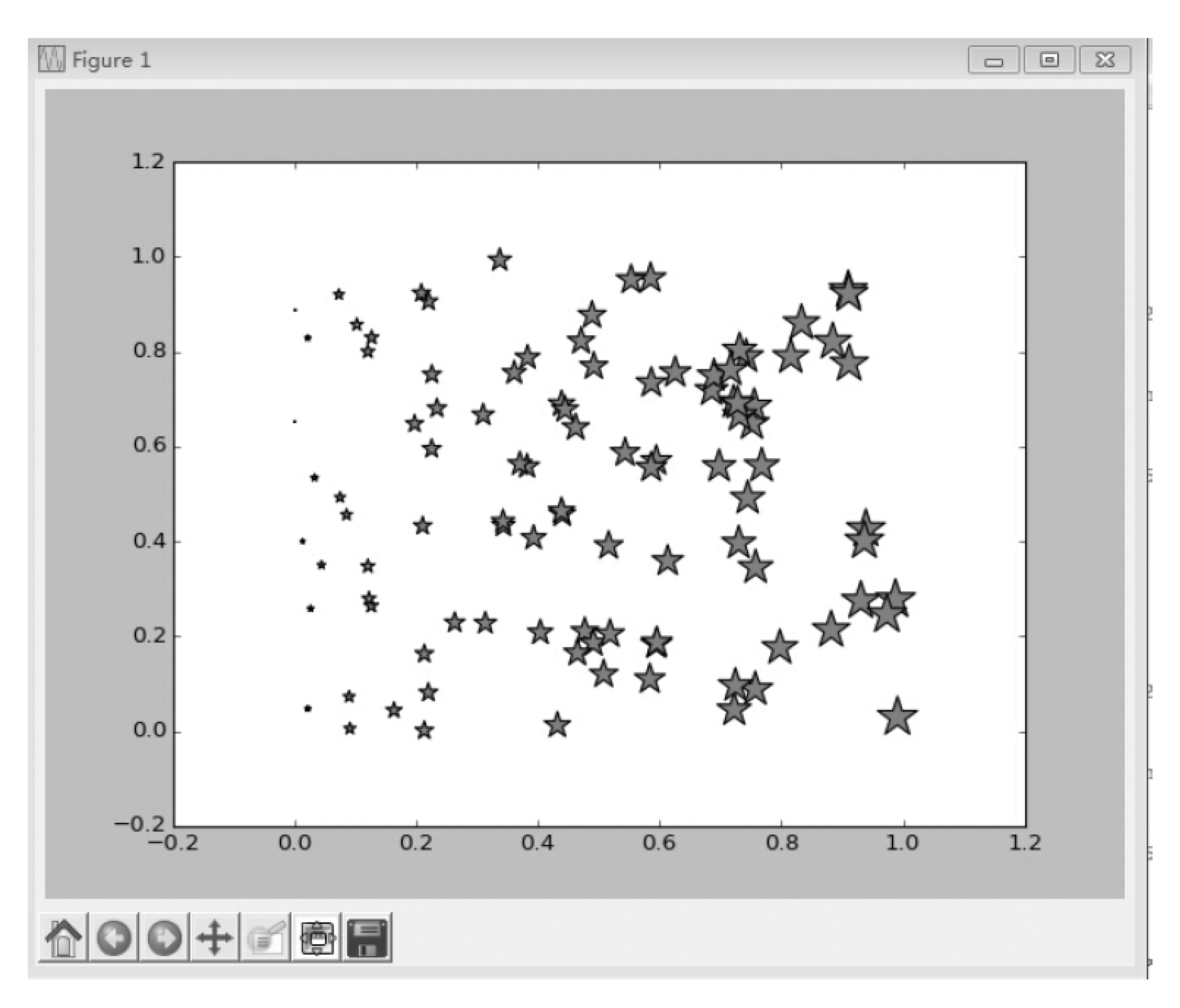


图 17-3　绘制星形散点图

```
plt.ylim(-1.2,1.2)
plt.legend()                                    #显示图示
plt.show()                                      #显示绘图窗口
```

上面的代码运行结果如图 17-4 所示。

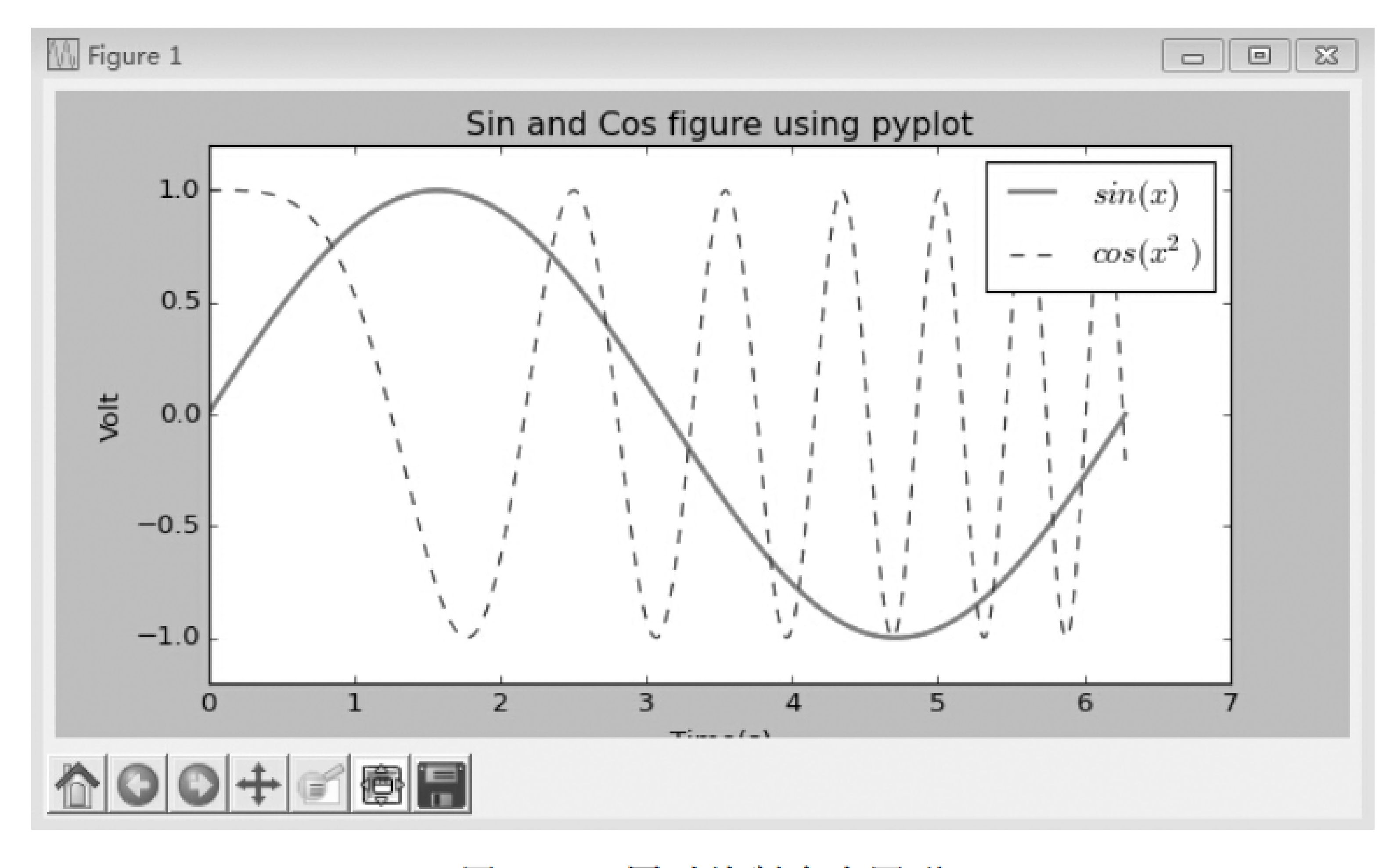


图 17-4　同时绘制多个图形

4. 使用 pyplot 绘制，多个图形单独显示

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x=np.linspace(0, 2* np.pi, 500)
y1 =np.sin(x)
y2 =np.cos(x)
y3 =np.sin(x* x)
#create figure
plt.figure(1)
#create three axes
#first line,first column
ax1 =plt.subplot(2,2,1)
#first line,second column
ax2 =plt.subplot(2,2,2)
#the whole second line
ax3 =plt.subplot(2,1,2)
#choose ax1
plt.sca(ax1)
#draw the curve in ax1
plt.plot(x,y1,color='red')
plt.ylim(-1.2,1.2)
#choose ax2
plt.sca(ax2)
plt.plot(x,y2,'b--')
plt.ylim(-1.2,1.2)
#choose ax3
plt.sca(ax3)
plt.plot(x,y3,'g--')
plt.ylim(-1.2,1.2)
plt.legend()
plt.show()
```

运行结果如图 17-5 所示。

5. 绘制三维图形

```
import numpy as np
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d
x,y =np.mgrid[-2:2:20j, -2:2:20j]
z =50 * np.sin(x+y)
ax =plt.subplot(111, projection='3d')
ax.plot_surface(x,y,z,rstride=2, cstride=1, cmap=plt.cm.Blues_r)
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```

运行结果如图 17-6 所示，在绘图窗口中可用鼠标来旋转所绘制图形：

下面的代码绘制了另一个略加复杂的三维图形：

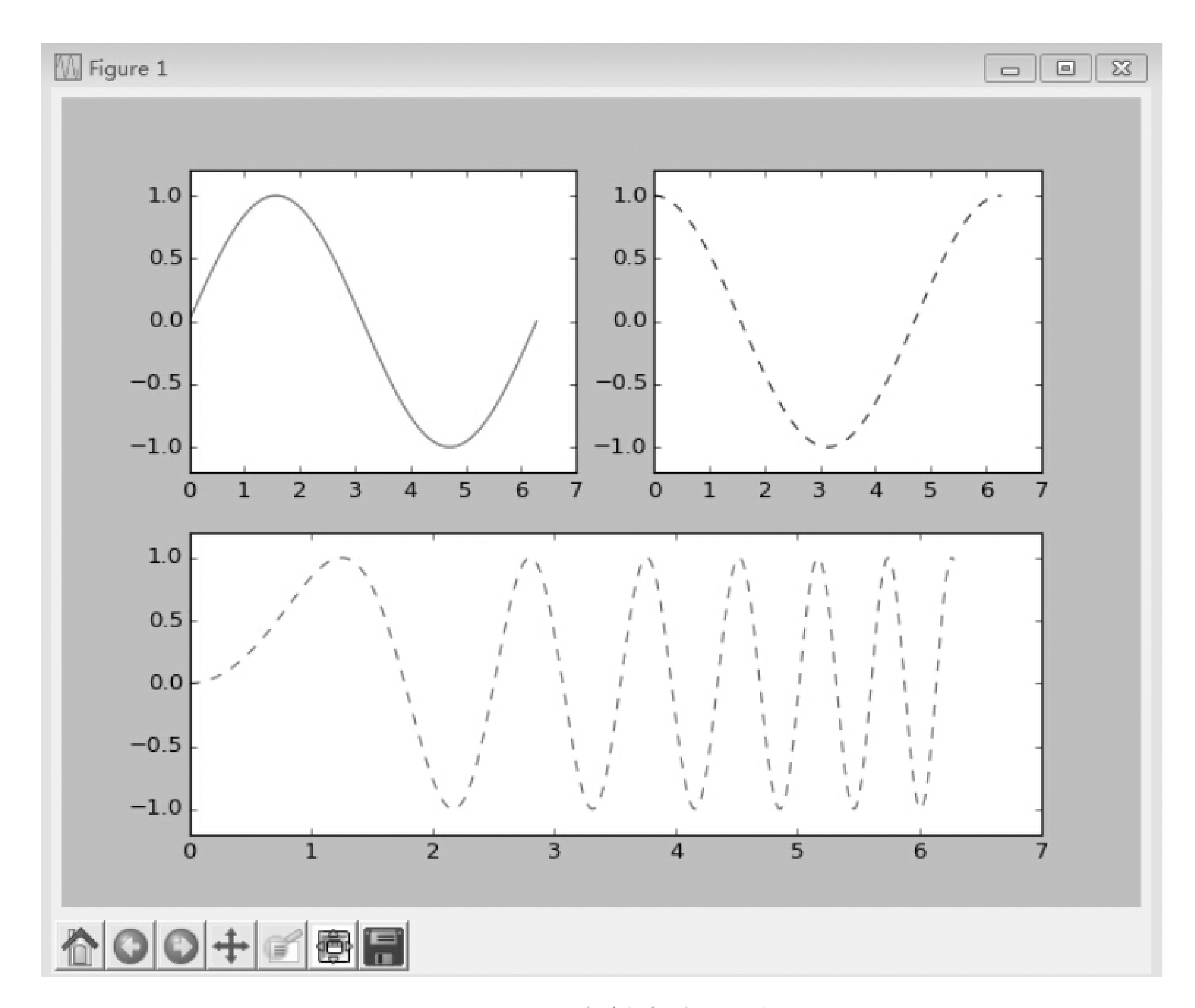


图 17-5　绘制多个图形

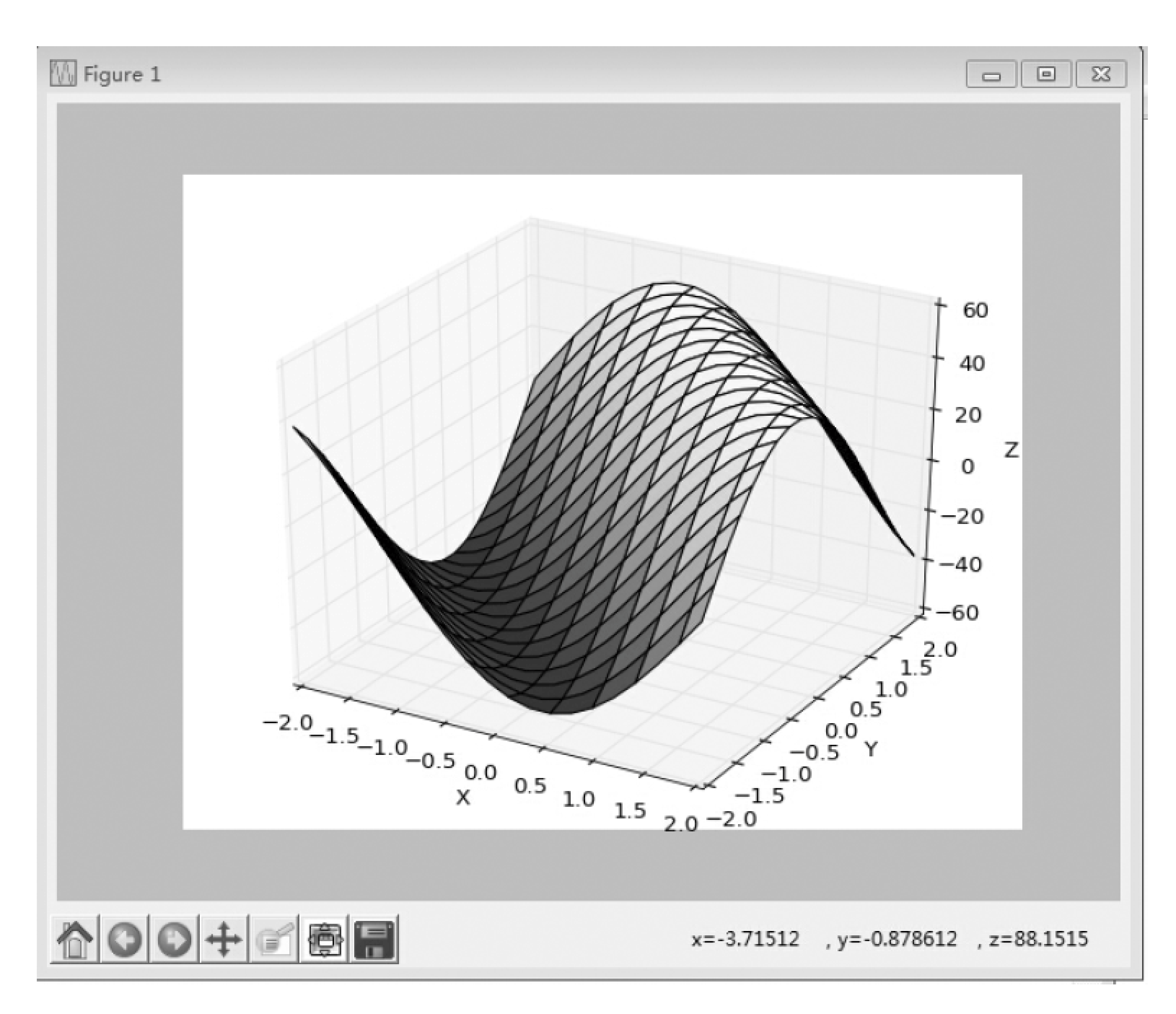


图 17-6　三维图形

```
import pylab as pl
import numpy as np
```

```
import mpl_toolkits.mplot3d
rho, theta = np.mgrid[0:1:40j, 0:2 * np.pi:40j]
z = rho**2
x = rho * np.cos(theta)
y = rho * np.sin(theta)
ax = pl.subplot(111, projection='3d')
ax.plot_surface(x,y,z)
pl.show()
```

运行结果如图 17-7 所示。

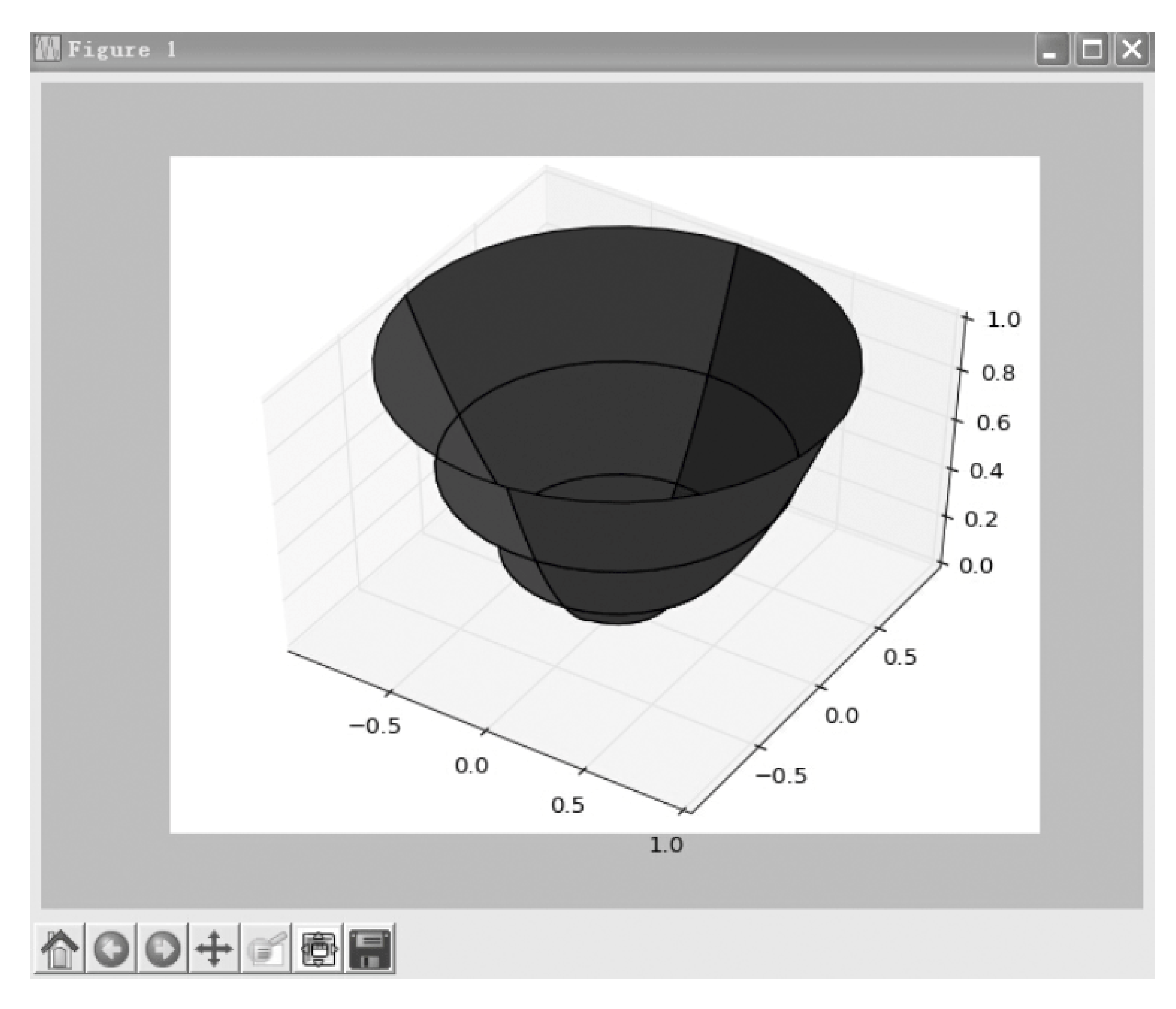


图 17-7 绘制三维图形

本章知识精要

（1）比较常用的科学计算可视化模块有 NumPy、SciPy 和 Matplotlib。

（2）NumPy 支持数组与标量的运算、数组与数组的运算、向量内积、数组的三角函数运算、数组多元素操作、不同维度的最大值与均值计算、切片操作、计算标准差与方差以及特殊数组生成等功能。

（3）SciPy 模块依赖于 NumPy 模块，在其基础上增加了大量用于数学计算、科学计算以及工程计算的模块，包括线性代数、常微分方程数值求解、信号处理、图像处理和稀疏矩阵等。

（4）Matplotlib 模块依赖于 NumPy 和 tkinter 模块，可以绘制多种形式的图形，包括线图、直方图、饼状图和散点图等，是科学计算可视化的重要工具。

习　　题

1. 运行本章所有代码并查看运行结果。

2. 使用 Python 内置函数 dir()查看 Scipy 模块中的对象与方法，并使用 Python 内置函数 help()查看其使用说明。

参考文献

[1] 张颖，赖勇浩. 编写高质量代码——改善 Python 程序的 91 个建议[M]. 北京：机械工业出版社，2014.

[2] 杨佩璐，宋强等. Python 宝典[M]. 北京：电子工业出版社，2014.

[3] 袁国忠译. Python 编程入门[M]. 北京：人民邮电出版社，2013.

[4] 张若愚. Python 科学计算[M]. 北京：清华大学出版社，2012.

[5] 赵家刚，狄光智，吕丹桔，等. 计算机编程导论——Python 程序设计[M]. 北京：人民邮电出版社，2013.

[6] Michael Hale Ligh，Steven Adair，Blake Hartstein，Matthew Richard. 恶意软件分析诀窍与工具箱——对抗“流氓”软件的技术与利器[M]. 胡乔林，钟读航译. 北京：清华大学出版社，2012.

[7] 李锐，李鹏，曲亚东，等译. 机器学习实战[M]. 北京：人民邮电出版社，2013.

[8] Wes Mckinney. 利用 Python 进行数据分析[M]. 唐学韬等译. 北京：机械工业出版社，2014.

[9] 张银奎. 调试软件[M]. 北京：电子工业出版社，2013.